Investigation in the Application of Complex Algorithms to Recurrent Generalized Neural Networks for Modeling Dynamic Systems

A Thesis Submitted to the College of Graduate Studies and Research

In Partial Fulfillment of the Requirements

For the Degree of Master of Science

In the Department of Mechanical Engineering

University of Saskatchewan

Saskatoon

By

R. Matthew Yackulic

PERMISSION TO USE

In presenting this thesis/dissertation in partial fulfillment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis/dissertation in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis/dissertation work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis/dissertation or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis/dissertation.

Requests for permission to copy or to make other uses of materials in this thesis/dissertation in whole or part should be addressed to:

Head of the Department of Mechanical Engineering
University of Saskatchewan
57 Campus Drive
Saskatoon, Saskatchewan
S7N 5A9
Canada

ABSTRACT

Neural networks are mathematical formulations that can be “trained” to perform certain functions. One particular application of these networks of interest in this thesis is to “model” a physical system using only input-output information. The physical system and the neural network are subjected to the same inputs. The neural network is then trained to produce an output which is the same as the physical system for any input. This neural network model so created is essentially a “blackbox” representation of the physical system. This approach has been used at the University of Saskatchewan to model a load sensing pump (a component which is used to create a constant flow rate independent of variations in pressure downstream of the pump). These studies have shown the versatility of neural networks for modeling dynamic and non-linear systems; however, these studies also indicated challenges associated with the morphology of neural networks and the algorithms to train them. These challenges were the motivation for this particular research.

Within the Fluid Power Research group at the University of Saskatchewan, a “global” objective of research in the area of load sensing pumps has been to apply dynamic neural networks (DNN) in the modeling of loads sensing systems.. To fulfill the global objective, recurrent generalized neural network (RGNN) morphology along with a non-gradient based training approach called the complex algorithm (CA) were chosen to train a load sensing pump neural network model. However, preliminary studies indicated that the combination of recurrent generalized neural networks and complex training proved ineffective for even second order single-input single-output (SISO) systems when the initial synaptic weights of the neural network were chosen at random.

Because of initial findings the focus of this research and its objectives shifted towards understanding the capabilities and limitations of recurrent generalized neural networks and non-gradient training (specifically the complex algorithm). To do so a second-order transfer function was considered from which an approximate recurrent generalized neural network representation was obtained. The network was tested under a

variety of initial weight intervals and the number of weights being optimized. A definite trend was noted in that as the initial values of the synaptic weights were set closer to the "exact" values calculated for the system, the robustness of the network and the chance of finding an acceptable solution increased. Two types of training signals were used in the study; step response and frequency based training. It was found that when step response and frequency based training were compared, step response training was shown to produce a more generalized network.

Another objective of this study was to compare the use of the CA to a proven non-gradient training method; the method chosen was genetic algorithm (GA) training. For the purposes of the studies conducted two modifications were done to the GA found in the literature. The most significant change was the assurance that the error would never increase during the training of RGNNs using the GA. This led to a collapse of the population around a specific point and limited its ability to obtain an accurate RGNN.

The results of the research performed produced four conclusions. First, the robustness of training RGNNs using the CA is dependent upon the initial population of weights. Second, when using GAs a specific algorithm must be chosen which will allow the calculation of new population weights to move freely but at the same time ensure a stable output from the RGNN. Third, when the GA used was compared to the CA, the CA produced more generalized RGNNs. And the fourth is based upon the results of training RGNNs using the CA and GA when step response and frequency based training data sets were used, networks trained using step response are more generalized in the majority of cases.

ACKNOWLEDGEMENTS

The author wishes to express his gratitude for the encouragement, guidance and financial support to his supervisors, Dr. Richard Burton and Dr. Greg Schoenau, during the course of preparing this thesis. Appreciation is also sincerely expressed to Mr. Doug Bitner for his guidance and assistance during the course of the conducted studies.

The author acknowledges the financial assistance in the form of a Graduate Teaching Fellowship Award, provided by the Department of Mechanical Engineering at the University of Saskatchewan.

The author would also like to give a heartfelt thank you to his mother, Dorothy Yackulic, and to his brother, Andrew Yackulic, for their encouragement and support during the preparation of this thesis. The author would also like to thank his father, Alan Yackulic, for inspiring him towards his chosen path. You will always be missed ROM.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

NOMENCLATURE

α	Reflection coefficient	
$\delta(k)$	Partial derivatives for error used in BP	
σ	Activation function	
μ	Learning rate	
a	upper boundary of weight range	
b	lower boundary of weight range	
c	Non-uniformity parameter	
e	Error at a specific time or time step	
E	Summation of least squared errors for batch training	
$E^{(i)}$	Batch training error of individual i	
$E(k)$	Least squared error	
E_{max}	Maximum error	
$f^{(i)}$	Fitness factor of individual i	
f_p	Fitness of population	
$f_{ratio}^{(i)}$	Fitness ratio of individual i	
G	Number of times a new population has been calculated	
G_{max}	Maximum number of generations during GA training	
I^A	Individual with the highest fitness factor in mating pair	
I^B	Individual with the lowest fitness factor in mating pair	
$Index$	Index matrix for mating using the GA	
k	Time instant	
m	number of variables being solved for or optimized	
M	Number of layers in a neural network	
n	number of individuals in the population	
N	Number of time steps in a training set	
$\hat{\Delta p}$	Estimate of Δp	[kPa]
Δp	Change in pressure	[kPa]
P_L	load pressure	[kPa]

P_S	supply pressure or upstream pressure in a load sensing pump	[kPa]
Q	flow rate	[gal/min]
$Rand$	Random population matrix	
$s(k)$	Neuron output	
t	Time	[sec]
Δt	length of time step (noted as dt in some figures)	[sec]
W	Weight connection matrix for RGNN	
$\overline{W}$	Centroid of the complex	
W_D	Dynamic weight connection matrices for RGNN	
W^{exact}	Matrix containing exact values of neural weights	
W^h	Individual with the highest error	
w_{ij}	Weight connection between output of neuron j to input of neuron i	
$w^{i(mutation)}$	Weight after mutation	
ΔW_k	Perturbation matrix	
W^l	Individual with the lower error	
ω_n	Natural frequency	[rad/sec]
W^r	Reflection individual	
W_S	Static weight connection matrices for RGNN	
W_{tent}	Tentative population matrix	
x	Output of system	
X	Input of system as a vector matrix	
x^k	Input at time instant k	
y	Output of system	
Y	Output of system as a vector matrix	
y_d	Desired output of network	
y^k	Output at time instant k	
z^{-1}	Time step delay	

ABBREVIATIONS

NN	Neural networks
SNN	Static neural networks
DNN	Dynamic neural networks
DNU	Dynamic neural unit
RGNN	Recurrent generalized neural network
CA	Complex algorithm
FFNN	Feed-forward neural network
FBDNN	Feedback dynamic neural network
FFDNN	Feed-forward dynamic neural network
PI	Proportional-integral
PID	Proportional-integral-derivative
MIMO	Multi-input multi-output
HON	Higher order network
FLNN	Functional link neural network
MFFNN	Multilayer feed-forward neural network
BP	Backpropagation
GA	Genetic Algorithm
SISO	Single-input single-output

Chapter 1: Introduction and Objectives

1.1 Project Background and Motivation

The study of fluid power systems is important as they are used in a variety of aeronautical, mining, oil and gas, and other heavy duty applications [Lamontagne, 2001][Wu, 2003][Li, 2007]. However, complications have arisen in creating working models for fluid power components due to their nonlinear and dynamic nature throughout a large operating range. In order to properly design a complete system, each component must have an analytical model which accurately characterizes its actions over the complete system operating range.

If the operating range of the hydraulic system is small, then an all encompassing model is relatively simple involving a few coefficients and characteristics describing such things as the orifice size and shape. However, if the operating range is large then creating an accurate model becomes more complicated. An example of a system designed to work over a large operating range is a load sensing pump. This system has received considerable attention by researchers at the University of Saskatchewan [Bitner, 1986] [Xu, 1997] [Lamontagne, 2001] [Li, 2007] [Wiens, 2008a][Wiens, 2008b] [Wu, 2003] and was the original system that was the underlying focus of this study.

A load sensing pump works on a basic principle; no matter what the change of pressure is at the dominant load, the flow rate to the dominant load delivered by the pump is controlled to be constant [Bitner, 1986]. This is done so by creating a constant pressure drop across a controlling orifice. A constant pressure drop will create a constant flow rate assuming that the properties of the fluid do not change [Merritt, 1967].

The load sensing pump is comprised of three main components; a pressure compensated pump, a controlling orifice, and a compensator spool. A diagram of a load sensing pump is shown in Figure 1.1. With reference to Figure 1.1, the differential

pressure across the controlling orifice is fed back to the compensator spool. If the load pressure P_L increases, the compensator spool moves to the right, porting fluid from the controlling spool to tank. This reduces controlling pressure which in turn increases the swash plate angle increasing the pump flow. As a result, the upstream pressure (P_s) increases until the desired pressure differential is accomplished. Thus the flow rate through the orifice is reestablished.

Because the upstream pressure, P_s, is set marginally higher than the load pressure (typically 10 *kPa*), the pressure drop across the load control valve is fairly small, hence energy losses can be minimized. In circuits in which a load sensing pump delivers flow to several loads, a series of "shuttle" valves sense the dominant load pressure and this pressure is fed back to the compensator. As such the pressure drop across the dominant loading control valve is minimized, but the same cannot be said for pressure drops across the other load valves.

Each component of the load sensing pump contains non-linear and dynamic behavior on their own. When the three components in Figure 1.1 are combined, these dynamic components and non-linearities become even more difficult to properly describe using one model.

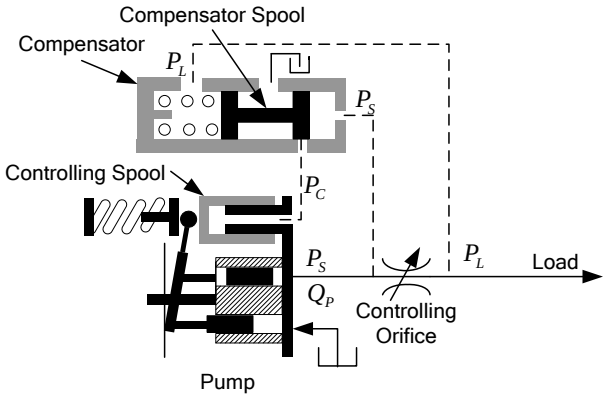


Figure 1.1: Schematic of a Load Sensing Pump [Li, 2007].

Linearized models based on fundamental fluid and component motion equations may be created for load sensing pumps; two such models were created by Bitner [1986]

and Krus [1988]. However an inherent disadvantage of linearized models is that they are only capable of working in a small operating range. For a load sensing pump to maintain a specific operating point, a variety of components may be arrayed in a variety of combinations to compensate for fluctuations downstream. These moving components will create multiple dynamic responses depending on not only the operating point of the pump, but also the internal operating points between the three components previously listed.

If the internal operating point is used as an input, then a model may be created which compensates for internal dynamics over a large internal and external operating range. This is a very difficult task to do analytically but the use of "black box" approaches, in which only input-output relationships are of interest, avoids these problems. One such method which has been considered to create this black box is the use of static and dynamic neural networks.

Neural networks (NN) are of two main types; static (SNN) and dynamic (DNN). Both types of networks are comprised of neurons connected by "synapses" which contain a weighting factor. The value of the weighting factor dictates the output of the NN. For modeling purposes a NN is trained using input and output data sets under which conditions the model is expected to operate. This data set of values is referred to as the training data. Details of how these NNs and their training algorithms work are discussed in Chapters 2 and 3 respectively.

NNs may be used to do a variety of processes including system control, parameter estimation and system modeling [Burton, 1999][Yu, 2005][Pollmeier, 2004][McNamara, 1997]. For the purposes of the following research NNs will be used for system modeling. Once training has been completed the model will be considered to be self sufficient and the neural weights may not be modified; this is referred to as simulation modeling.

1.2 Previous Research of Load Sensing Pumps at the University of Saskatchewan

As mentioned, there has been a variety of studies completed at the University of Saskatchewan which considered the modeling of load sensing pumps. The task was initially investigated by Bitner [1986], who considered the methodology of creating a linearized model for a load sensing pump. Bitner found the model degraded as the actual pump operating point diverged from the model designed operating point. To overcome this drawback a variety of operating points may be considered and the parameter estimation could be created for each operating point. This would create a set of linearized equations for each operating point being considered. Although achievable, this process would be very labour intensive and time consuming due to the large operating range of a load sensing pump.

One option to negate the disadvantages of linearized models is to use the concepts of "black box" and "gray box" designs. A black box model works on the principle that the user does not concern themselves with system or component properties, but rather cares solely about the input-output relationship [Li, 2007]. When creating a black box model, there is no intent to create a model which fits specific parameters of a component. A variation on a black box model is a gray box model; a gray box model takes into consideration the system parameters. One such example would be taking a mathematical model which is known to describe one step in a large component, and combining it with a black box model which describes the remainder of the component [McNamara, 1997].

To overcome the drawbacks of the linearized model, Xu [1997] used the concept of NNs to create an experimental model of a load sensing system. The architecture chosen by Xu was a partially recurrent neural network with delayed inputs which were used as the inputs to a multilayer feed-forward network with a hyperbolic tangent function as the hidden layer output function. The network also used the conjugate gradient algorithm for training.

Xu found that although the method worked well to create a "universal approximator" [Xu, 1997] there were still some drawbacks. The main drawback was the error accumulation which occurred as the simulation time increased once the model was created. The error was attributed to the quality of the experimental data used for training. It was recommended that the richness of data, with respect to frequency and magnitude, should be considered and improved upon.

Xu's research in the area of modeling load sensing pumps was extended by Lamontagne [2001]. A major consideration that Lamontagne made was which input-output data relationship would work the best. He considered two different morphologies; a separate pump and valve combination, and a stand-alone pump model. He found that a separate pump model using NNs may be created; however a known valve model must be obtained which can be very difficult using the experimental relationship between supply pressure (P_S) and the flow (Q).

The morphology of the NN was also changed to try and decrease the error accumulation. In order to do this a feed-forward network with input time delays was used. The delaying of the input created the dynamic effects needed to properly model the system; however it negated the dependence on the output of the system which created the error accumulation. Lamontagne showed that Xu's theory of a richer training set was correct. It was found that higher quality data in both frequency and magnitude produced more accurate results, especially in the high magnitude ranges of non-linear systems.

The next research conducted used the concepts of dynamic neural units (DNUs), this was done by Li [2007]. DNUs use time delayed feed-forward and recurrent connections to create the dynamic effects. However unlike other NN morphologies all of the connections are summed up at the end of the neuron instead of at the end of the network. This helps to minimize the error accumulation caused by the network as long as an appropriate neural pattern is picked such as a second order system which does not rely on the gradient of the function.

Li also studied the different types of models being used; a pump/compensator/valve model, a pump/compensator model, and a pump only model. It was found that the pump only model avoided problems such as data quality and input independence better than the other two models. However a significant steady state error arose when the pump only model was created using the chosen DNU/DNN structure. Because the steady state error is not a dynamic error, a simple SNN was put in series with the DNU/DNN structure in order to compensate. Although effective this addition created the need for two separate training sequences for both dynamic and static response, which resulted in an increase in training time.

Most recently while working on a neural fuel control system, Wiens [2008a] used a network structure called a recurrent generalized neural network (RGNN) along with a complex training algorithm (CA). The morphology of RGNNs and the concepts of the complex training algorithm will be explained in more detail later in this thesis. Using the training data created by Lamontagne and also used by Li, Wiens trained a DNN to model a load sensing pump. When compared to earlier research it was found that the output data created by the model after training was significantly better than that of Xu, Lamontagne and Li while only using the single network. However, in order for the RGNN to be trained, extensive "tuning" of the network was necessary before a satisfactory result could be accomplished.

Initial studies were completed during the creation of a CA program similar to that used by Wiens. It was apparent that using the RGNN as a true black box – no preconceived knowledge of the system being modeled – would not produce satisfactory results. The basis to justify the achievement of satisfactory results will be discussed in Chapter 4. It became clear that a better understanding of the limitations of the RGNN trained by the CA was necessary, and hence was the motivation of this thesis.

1.3 Thesis Objectives

The "global" objective of research in the area of load sensing pumps using neural networks is to create a DNN based computer model. However, because of difficulties in training of a special form of a DNN (the recurrent generalized neural network (RGNN)), the research focus shifted towards an objective of providing a better understanding of the limitations of the DNN and the training method used, in particular, by Wiens in his preliminary studies. It is expected that the RGNN model should be able to take appropriate pressure inputs and create a flow rate output which agrees with experimental values obtained. This computer model should be able to link to other fluid power components using a single software platform such as MATLAB.

The prime objective was to study the competency of using RGNNs along with the complex training method which was used by Wiens [2008b]; the complex training method is a non-gradient which will discussed further in Chapter 3. The second objective was to complete a comparative study of the complex training method to a heuristic genetic algorithm, which is also a non-gradient training method, using RGNNs. In using the genetic training method which has been shown to work in many other applications, studies can be done to discover whether any issues that arose with using the complex method were the consequence of an inadequate training algorithm or network topology.

1.4 Outline and Structure of Thesis

In Chapter 2 NNs will be explained more in depth with the main focus being on dynamic neural networks and the advantages and shortcomings of different types of networks.

Chapter 3 will discuss the two main training method types; gradient and non-gradient, and will give examples for each. Non-gradient training methods will be introduced and their main advantages will be outlined for creating DNNs.

Chapter 4 will study the results of applying the complex method to a RGNN using a variety of inputs and weight initialization methods. The effects of small-perturbations from an exact representation of a system discussed earlier will be studied to observe the effectiveness of the complex method.

Chapter 5 will incorporate the use of another non-gradient training method in order to create a benchmark for the complex method to be compared to. The training method considered was a heuristic genetic algorithm; a RGNN will be trained under the same conditions as the complex method used in Chapter 4.

The final chapter, Chapter 6, will be used to summarize the findings, give conclusions based on those findings, and to discuss recommended future research involving the project.

Chapter 2: Dynamic Neural Networks and Applications to Fluid Power Systems

2.1 Introduction

The objective of this Chapter is to present basic information on neural networks. In addition, because the overall goal of research in this area was focused on fluid power systems, several examples to demonstrate how static and dynamic neural networks have been used are considered.

As discussed in Chapter 1, neural networks are a very useful tool for both static and dynamic applications. Before a network can be constructed first the type of system being analyzed must be studied. The main consideration is to distinguish between what type of output is being provided by the system; static or dynamic. The following section will give a brief explanation of static neural networks, but mainly deal with considerations which must be made for dynamic systems.

2.2 Static Neural Networks

Static neural networks (SNNs) are used for input-output relationships which have no derivative or integral functions, and therefore contain no dynamics. Although SNNs are not useful for dynamic operations – which limit their application for the modeling of fluid power systems – they are highly useful for non-linear systems static systems [Gupta, 2003][Haykin, 1999]. Figure 2.1 shows a feed-forward neural network (FFNN) which is a common static neural network type.

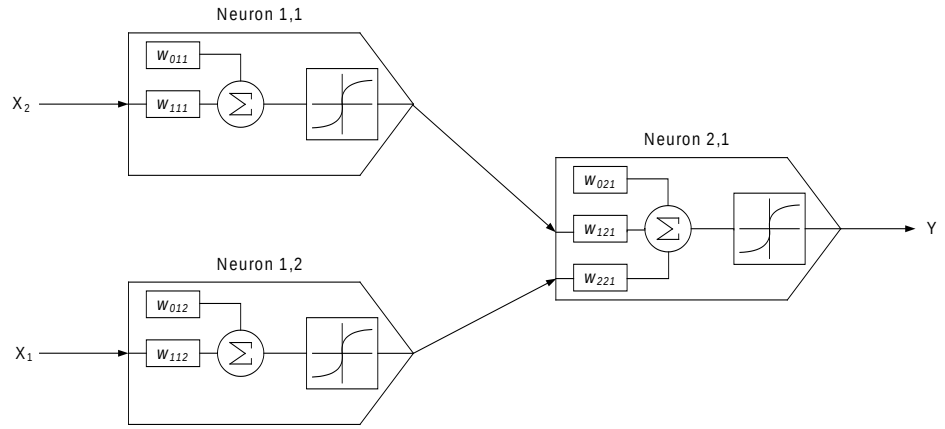


Figure 2.1: Schematic of example static network consisting of two inputs and one output.

All neural networks are formed with the use of neurons in series or parallel with a variety of connection types. SNNs contain no time-delayed connections in either the interior of the neuron or in any of the exterior connections. Figure 2.2 shows a labeled static neuron schematic of which typical SNNs are comprised. Static neurons are comprised of three main components; (a) synaptic weights (essentially gains), (b) a summation junction, and (c) an activation function, denoted by σ in Figure 2.2. As will be discussed in Chapter 3, the training of a neural network is completed by optimizing the value of the synaptic weights such that the output from the network approaches some desired output. As many inputs as necessary are sent into the neuron and each input along with a bias, passes through a synaptic weight. The purpose of the bias is to help the neural network maintain stability [Gupta, 2003].

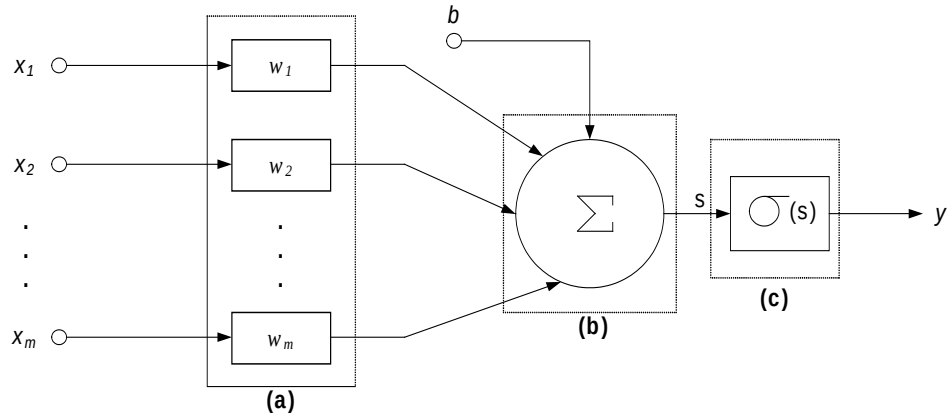


Figure 2.2: Schematic of a static neuron [Haykin, 1999].

The summation junction receives the outputs from the synaptic weights and adds all the incoming signals to the neuron. Once the signals are added, the summation signal is sent through an activation function; the activation function is the main component which allows the use of neural networks for nonlinear applications. A number of types of activation functions can be used, but for hidden layers the most common type is the hyperbolic tangent function. This nonlinear function places a limit of negative one to plus one; however in-between these values, the function is nonlinear. If the output ranges between negative one and plus one then a hyperbolic tangent function may be used. However, because the hyperbolic tangent function has a limit it is not well suited as the activation function of the output neuron. FFNN can be comprised of any number of connections as long as no time delays are present, and can be used to model both linear and nonlinear systems.

2.3 Types of Dynamic Networks

Neural networks have many different types of networks, and each type of network can branch off into a large number of subtypes. Each type of network has its advantages and disadvantages which must be considered when choosing an archetype. The most common dynamic neural network form are feedback dynamic neural networks (FBDNNs) [Haykin, 1999][Gupta, 2003][Xu,1997]. A FBDNN uses multiple delayed feedbacks to create a dynamic output for the system being considered.

As mentioned each archetype of network can have a large variation of their application. Figure 2.3 shows the most simple of applications where the output of the system is delayed by one time step and is then fed back to the system as an input. Similarly the outputs may be fed back to intermediate layers instead of being used as an input, or intermediate layer outputs can be fed back instead of the system output [Gupta, 2003].

One of the main drawbacks to the FBDNN is the propagation of system error once a system is disengaged from the training process [Xu, 1997]. This is because the system

output is fed back repeatedly, and subsequently if the training process has any error then the error is fed back repeatedly. The longer a system runs without resetting itself the larger the error propagation will become.

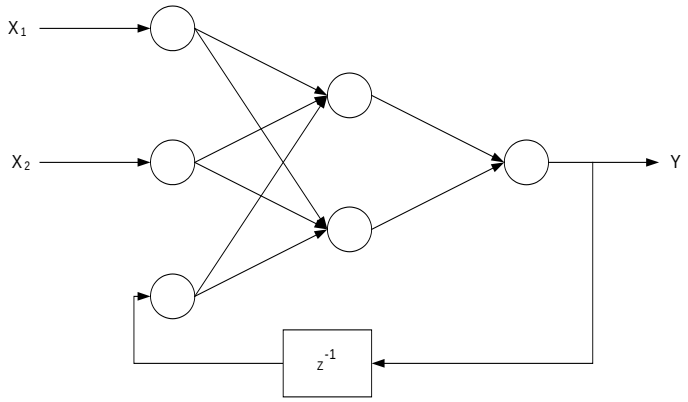


Figure 2.3: Schematic of feedback dynamic neural network (FBDNN) archetype [Haykin, 1999].

Because of issues such as error accumulation [Xu, 1997] other neural network patterns and system methodologies have been created to overcome these setbacks. For example, by limiting the dynamics of the system to a single layer rather than to the entire network output, the dependence of the dynamic output of the system on error accumulation becomes less apparent.

Another type of neural network archetype is a feed-forward dynamic neural network (FFDNN) shown in Figure 2.4. Unlike the FBDNN, the FFDNN uses time delays in a feed-forward orientation to create system dynamics. Figure 2.4 shows a FFDNN which uses delayed inputs to the system at multiple layers.

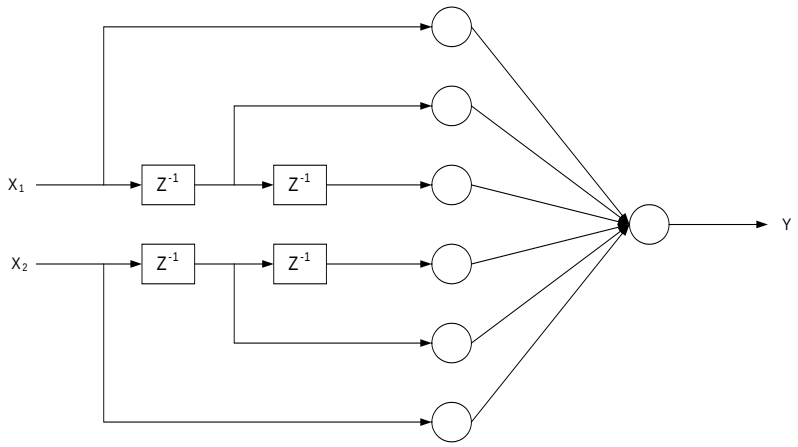


Figure 2.4: Schematic of feed-forward dynamic neural network (FFDNN) archetype [Lamontagne, 2001].

In a regular feed-forward neural network there are no delayed paths; this type of network is used for static cases. However, in a dynamic case if the input has been delayed at multiple time steps then the effect of the input does not become apparent initially, but manifest itself in a delayed fashion. Because the output is not being fed back to create the dynamic aspects of the system, the error is not continually fed back to either the input of the system or the inner layers. This omission of error propagation alleviates the problem of the error becoming time dependent.

Although in FFDNNs the error propagation problem is alleviated, the concern with error and system performance is still present. In classical control systems with a negative feedback the steady state error will be reduced using the feedback line and continue to do so until an acceptable error is present. Once disconnected from the training mechanism FFDNNs do not have any sense of what is occurring at the output. So, although a steady state error may be present the network assumes that delayed inputs should be sufficient to minimize the steady state error. However, this is not always the case, since the lack of output feedback can lead to errors when the network is disconnected from the training process.

In order to apply a FFDNN properly, either a very large number of feed-forward paths must be present, which can have large costs in terms of computing power, or more

properties must be known about the system. If characteristics such as the maximum settling time in the system can be obtained then the correct number of delayed paths can be obtained. However finding such properties can be very difficult to obtain and this would involve knowing more about the system, changing the black box approach of neural networks to a grey box approach.

Newer approaches to neural network archetype design implement the concepts of both feed-forward and feedback system delays. One such archetype involves the use of Dynamic Neural Units (DNUs). Although a DNU network can contain state feedback signals from the output or other neurons, DNUs can contain all of the dynamic aspects of the network inside of one neuron. An example of a DNU is shown in Figure 2.5 and its application to a network Figure 2.6.

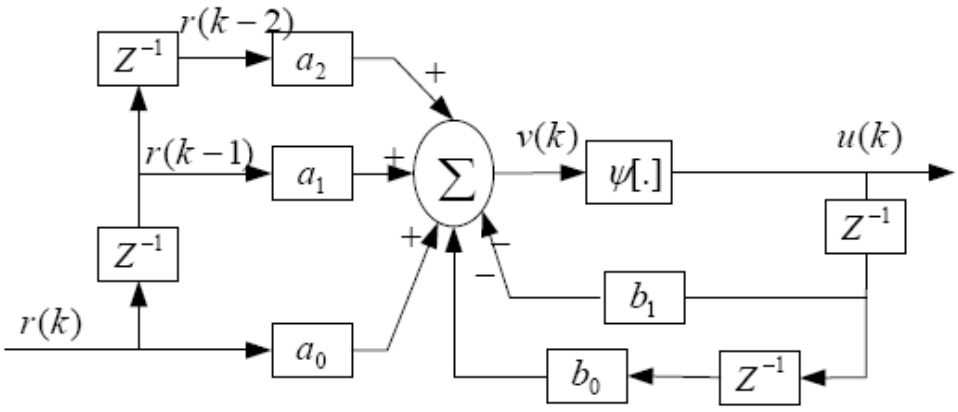


Figure 2.5: Schematic of a dynamic neural unit (DNU) architecture [Srivastava, 1998].

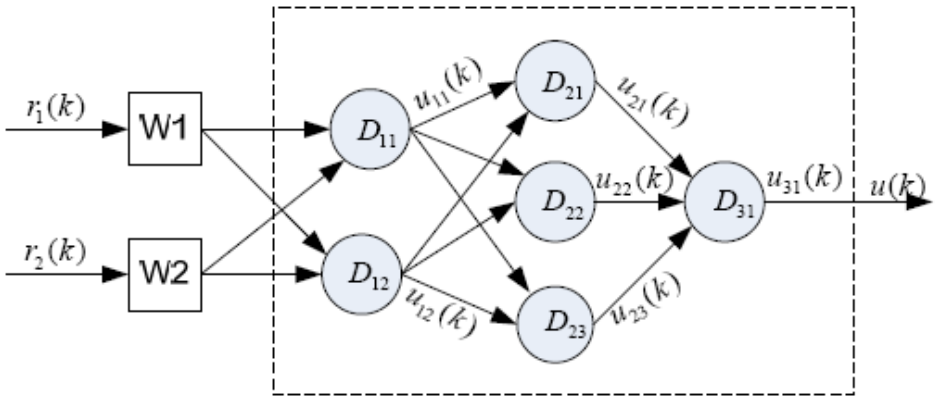


Figure 2.6: Structure of a three stage dynamic neural network (DNN) using six DNUs [Li, 2007]

The main advantage of using DNUs is the same as FFDNN; they do not require output feedback to create a dynamic response. DNUs only require the state feedback signals from the neuron and the other neurons around it. Because of this, there is no dependence on the overall output of the system. Also, unlike both the FFDNN and FBDNN, the dynamic contributions happen at each stage of the network rather than simply creating a delayed input or feeding back the output.

2.4 Recurrent Generalized Neural Networks

Another newer approach to neural networks configuration is the combination of feed-forward and feedback neural networks. Recurrent generalized neural networks (RGNNs) use connections between each neuron, both in the forward and backward directions, and each connection can have time delays depending on what the user specifies. The general form of a RGNN using delayed negative feedback lines is shown in Figure 2.7. It should be noted that any lines, including feed-forward lines, may contain time delays depending on how the user wishes to setup the network. The RGNN is a sub form of generalized neural networks where all neurons are connected to each other [Werbos, 1990]; but in the case of RGNNs, the recurrent connections are time delayed.

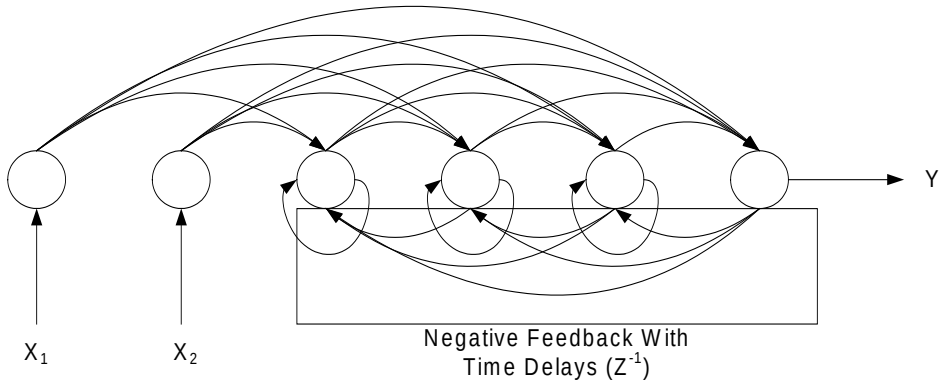


Figure 2.7: Recurrent dynamic generalized neural network (RDGNN) archetype [Wiens, 2008b].

Given the network shown in Figure 2.7 the output equation can be written as follows,

$$Y^k = W_S X^k - W_D Y^{k-1}, \tag{2.1}$$

where k refers to the time instant being considered, W_S is the weighting matrix for the feed-forward static connections, W_D is the weighting matrices for the feedback dynamic connections, X_1 and X_2 are the inputs to the network, and Y is the output of the network. The static weight matrix is given by the lower triangular matrix,

$$W_S = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ w_{21} & 0 & 0 & 0 & 0 & 0 \\ w_{31} & w_{32} & 0 & 0 & 0 & 0 \\ w_{41} & w_{42} & w_{43} & 0 & 0 & 0 \\ w_{51} & w_{52} & w_{53} & w_{54} & 0 & 0 \\ w_{61} & w_{62} & w_{63} & w_{64} & w_{65} & 0 \end{bmatrix}, \tag{2.2}$$

where the subscripts in the weight, w_{21} refers to the weight connection from the input neuron denoted by 1 to the first hidden neuron denoted by 2. The dynamic weight matrix is given by the upper triangular matrix,

$$W_D = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & w_{22} & w_{23} & w_{24} & w_{25} & w_{26} \\ 0 & 0 & w_{33} & w_{34} & w_{35} & w_{36} \\ 0 & 0 & 0 & w_{44} & w_{45} & w_{46} \\ 0 & 0 & 0 & 0 & w_{55} & w_{56} \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}. \quad \textbf{(2.3)}$$

Note that in Equation 2.3 the first row and the bottom right cell are all zero. This occurs because these connections do not exist. For the first row, all of the values are weights in the feedback lines to neuron one; the first neuron is the input neuron, therefore none of the outputs of any neurons will be fed to the input resulting in zeroes for this row. The bottom right cell is the output neuron; it is zero because, unlike the hidden layers, the output neuron only feeds back to other neurons; it does not feedback a delayed signal to itself.

Any one of the neural network archetypes mentioned above can be represented by a generalized neural network topology; an example is given in Figure 2.8. Figure 2.8 shows the FBDNN that was shown in Figure 2.3 using a generalized archetype. In a FBDNN each layer can contain a varying number of neurons and feed into either previous layers or subsequent layers. However, in a RGNN each layer has only a single neuron but those layers can connect to any other layer. This means that unlike a regular multilayer dynamic network the dynamic capabilities can come from within the layer.

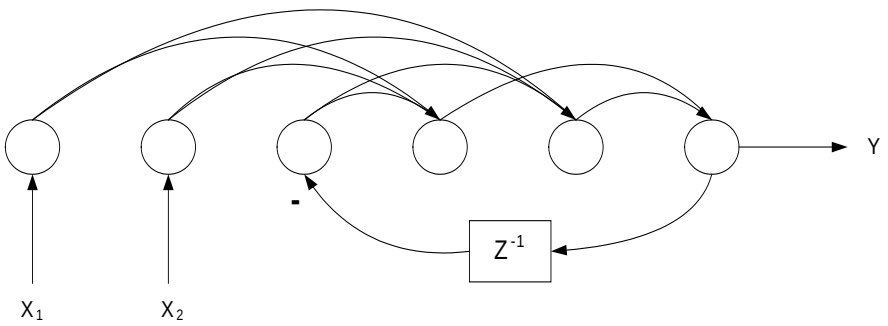


Figure 2.8: FBDNN in RGNN form.

Not only can RGNNs be used to represent other neural network archetypes, but they can also be used to represent mathematical equations directly. One such example is the representation of transfer functions in control systems. To illustrate RGNN capability, consider the stable linear dynamic transfer function,

$$\frac{Y(s)}{X(s)} = \frac{s+4}{s^2+2s+9}. \tag{2.4}$$

Taking the inverse Laplace transform of Equation 2.4 while assuming zero initial conditions gives,

$$\frac{d^2y}{dt^2} + 2\frac{dy}{dt} + 9y = \frac{dx}{dt} + 4x. \tag{2.5}$$

If it is assumed that $\Delta t \rightarrow 0$, then Equation 2.5 can be discretized into the following,

$$\frac{y^k - 2y^{k-1} + y^{k-2}}{\Delta t^2} + \frac{2y^k - 2y^{k-1}}{\Delta t} + 9y^k = \frac{x^k - x^{k-1}}{\Delta t} + 4x^k. \tag{2.6}$$

After combining like terms and noticing that the input needs a one-step time delay while the output needs a two-step time delay, Equation 2.6 may be represented using Figure 2.9.

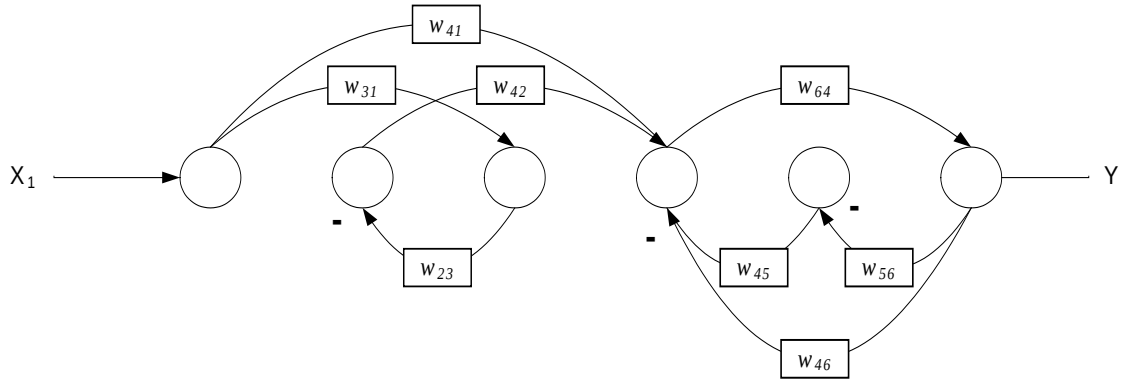


Figure 2.9: RGNN representation of transfer function given in Equation 2.4.

Unlike Equation 2.1, which calculates the output for a neural network with two inputs, the transfer function given in Equation 2.4 only uses one input. For a single-input single-output (SISO) RGNN,

$$Y^k = W_S X^k - W_D Y^{k-1}. \tag{2.7}$$

Because there is only one static and one dynamic weighting matrix which are represented by lower and upper matrices respectively, the weighting matrix for Figure 2.9 is,

$$W = W_S + W_D$$

$$W = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & w_{23} & 0 & 0 & 0 \\ w_{31} & 0 & 0 & 0 & 0 & 0 \\ w_{41} & w_{42} & 0 & 0 & w_{45} & w_{46} \\ 0 & 0 & 0 & 0 & 0 & w_{56} \\ 0 & 0 & 0 & w_{64} & 0 & 0 \end{bmatrix}. \qquad \textbf{(2.8)}$$

The accuracy of the RGNN output of Equation 2.4 is dependent on the Δt chosen for the system. If a large Δt is chosen then the error of the system becomes very large; conversely as Δt approaches zero the error between the continuous system output and the RGNN output approaches zero. Although the error between the actual system and the RGNN approaches zero, a very small Δt is not computationally efficient for training a neural network.

Figure 2.10 shows the RGNN output for Δt's of 0.005 seconds and 0.02 seconds for a step input. It should be noted that in Figure 2.10, the time step (Δt) is represented by dt due to the capabilities of the program used for the creation of the figure. Although the 0.02 seconds time step shows a larger error, it is still useful for testing the capabilities of a neural network; the concept of using time steps which are not sufficiently small will be discussed further in Chapter 4.

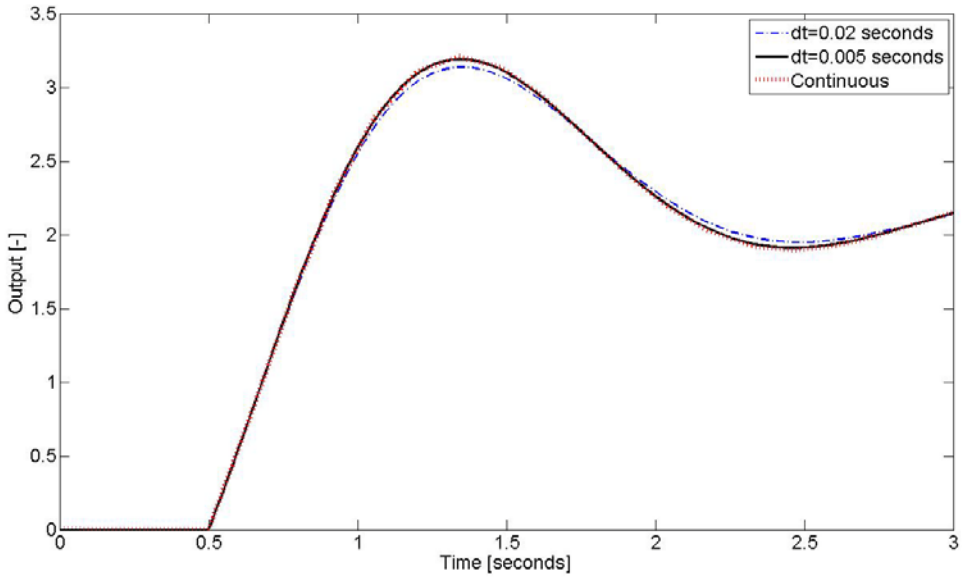


Figure 2.10: Output for a continuous transfer function compared to RGNN outputs at varying Δt.

2.5 Applications of Neural Networks to Fluid Power

As discussed in Chapter 1, fluid power systems are governed by nonlinearities and dynamics. Because of this the use of neural networks as a tool for studying fluid power systems has increased for a variety of purposes. The applications of neural networks can be broken down into three main categories; controlling, condition monitoring and simulating fluid power systems. The following section will outline research completed in the field of neural networks and fluid power for each of the three categories.

2.5.1 Neural Network Control Applications to Fluid Power

A main contributor to the field of neural network applications to fluid power systems has been the Fluid Power and Research Group at the University of Saskatchewan. Burton et al [1999] summarized three separate studies that were completed using neural network control for different types of systems. The first system

studied was a velocity-controlled rotary servo-system; the main objective was to reproduce a low-frequency sinusoidal angular velocity pattern [Burton, 1993]. Two classical controllers were used to create a benchmark; a proportional controller and a proportional-integral (PI) controller.

In order to improve on the performance of the classical controllers a neural network controller was created and trained using seventy-five training pairs. Once the training was completed (in this case the training compliance was based upon the achievement of a specific error), the neural network controller showed a deterioration in accuracy as the frequency increased. However, using the same testing sets the neural controller showed a higher degree of accuracy than both classical controllers.

For the second test a multi-gain proportional-integral-derivative (PID) controller was created to control a hydraulic servo-valve and linear actuator with non-linear friction [Qian, 1998]. Classical PID controllers can experience difficulty when being adapted to non-linear systems. Because of this draw back, PID controllers were created for seven different variations of the system which ranged across the different operating conditions. In essence, the PID controller would operate on the basis of a look up table with the ability to interpolate linearly between operating conditions.

To study the possible improvements over the multi-gain PID controller a recurrent multilayer feed-forward network was created using six different inputs ($\int e\,dt$, $e(k\text{-}1)$, $e(k\text{-}2)$, $e(k\text{-}3)$, $x(k)$, $x(k\text{-}1)$) and the velocity control signal as the output. The neural controller was trained to mimic the performance of the PID controller as shown in Figure 2.11. The training data used was created for the operating conditions for which the PID look up table was designed. Thus any incapability shown by the PID controller outside of the design range would not transfer to the neural network during the training process.

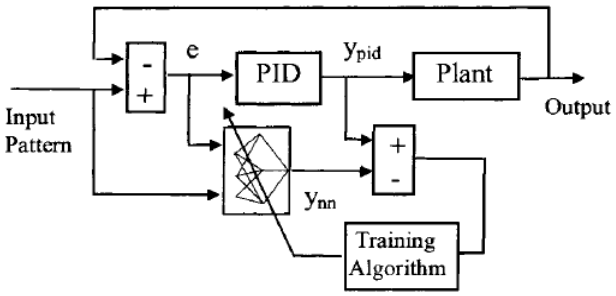


Figure 2.11: Training of a neural network to mimic the performance of a multi-gain PID controller [Burton, 1999][Qian, 1998].

Both the PID and neural controllers showed excellent accuracy when tested at the design points using a simulation model. However, when the input signal strayed from the design data, the PID controller accuracy decreased. This was most noticeable at low frequency inputs for which the controller was not designed. The neural network did not have the same problems at low frequencies, but it did show a small disturbance at the zero velocity condition.

To improve the performance of the neural controller a "kicker" was implemented externally to the controller as shown in Figure 2.12. The purpose of the "kicker" was to engage a pulse when the zero velocity condition was reached and was turned off when the output signal moved away from zero velocity. Although this proved to be effective the addition of the "kicker" compromised the concept of a black box. Because it was necessary to know that the system required the "kicker" to alleviate the zero velocity error, the neural network becomes a grey box by adding a second stage to the controller.

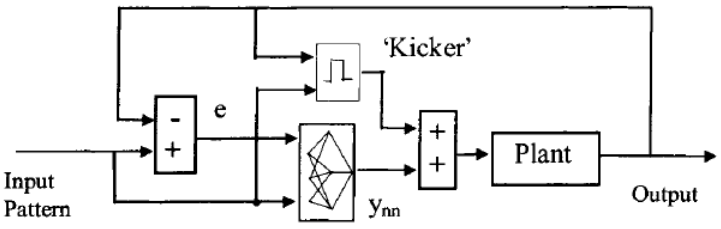


Figure 2.12: Neural controller mimicking multi-gain PID with external "kicker" [Burton 1999][Qian, 1998].

For the final test a neural controller was created to control a system which contained a position controller servo-system coupled to a force controlled servo-system [Zhang, 1996]. This created the necessity for a multi-input multi-output (MIMO) controller which took into account not only the input signal but also the movement of the coupled system. A desired output reference model was created using the Jacobian for the plant in order to obtain an output error which could be used to train the neural controller. A schematic of the neural network training scheme is shown in Figure 2.13.

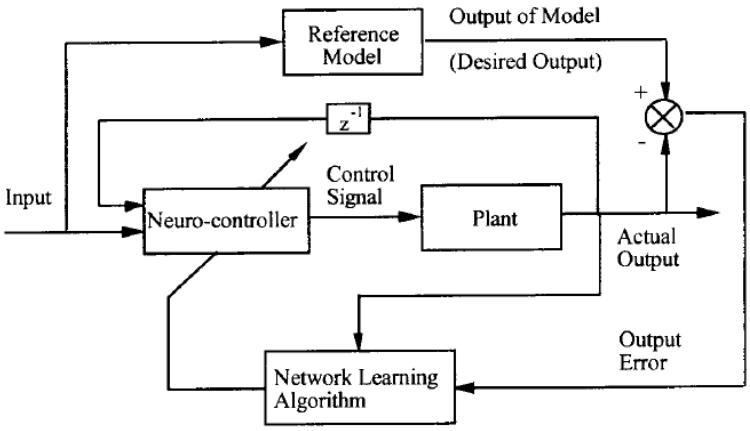
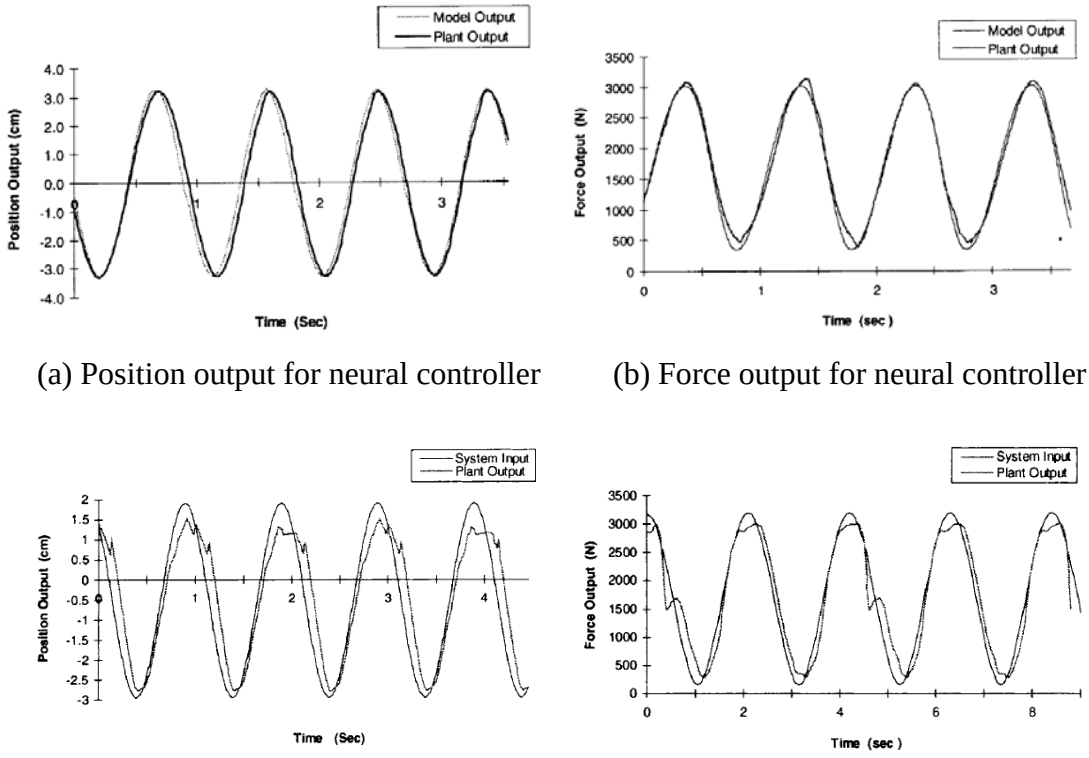


Figure 2.13: Neural network training schematic for coupled MIMO hydraulic system [Burton 1999][Zhang, 1996].

A three layer dynamic neural network was chosen which consisted of eight input neurons with both the servo inputs for force and position being delayed three time steps. The hidden layer consisted of eight neurons and the output layer had two neurons; one neuron for position output and one for force. Once the training process had taken place the neural controller was tested using a 1*Hz* sinusoidal input. In addition to the neural controller, a PID controller was designed for the system. Figure 2.14 shows the output signals for tests done at 1*Hz* for both the neural and PID controller. It can be seen that the neural controller performs superior to the PID controller although it should be noted that the force output does not track the reference model with the same accuracy as the position output.

As the frequency increased, both the neural and PID controllers began to deteriorate but the neural controller was consistently more robust then its PID counterpart. For all three tests where a neural controller was compared to a classical controller, the neural controller proved to be superior in performance.



(a) Position output for neural controller (b) Force output for neural controller

(c) Position output for PID controller (d) Force output for PID controller

Figure 2.14: Output profiles for MIMO hydraulic system using a neural controller (a, b) and PID controller (c, d) [Burton 1999][Zhang, 1996].

2.5.2 Condition Monitoring of Fluid Power Systems Using Neural Networks

There are a variety of ways to monitor the performance of fluid power systems such as using flow and pressure measurement devices. But as mentioned in Chapter 1, using a device such as an orifice results in unwanted pressure drops unrelated to the

systems performance. Researchers at Napier University [Yu, 2005] have studied a non-intrusive pressure measurement technique to overcome the drawbacks of pressure loss.

To measure pressure in hydraulic lines sound waves were used as fluctuations in pressure are proportional to the speed of sound in a fluid. Using a probe, a sound pulse was sent through a hydraulic line which reflected off the hose-wall on the opposite side and returned to the probe as shown in Figure 2.15. The pulse also penetrated the wall on the other side; both measurements were taken and used to calculate the speed of sound through the fluid.

Image has been removed temporarily due to Copyright

Figure 2.15: Schematic diagram of the non-intrusive pressure measurement [Yu, 2005].

A neural network was implemented to obtain the pressure using the relationship between the speed of sound and the fluid pressure. The network used was a functional link neural network (FLNN) which belongs to the higher order network (HON) family. HONs do not only use simple gain connections and summation junctions in their neurons, each neuron can be a higher order function. Therefore, an input could be squared or cubed before heading to a summation junction. In order to apply a FLNN some analytical knowledge of the network is necessary. For the non-intrusive pressure measurement, this is the relation between the pressure and speed of sound given by,

$$\Delta \hat{p} = \sum_{j=1}^{3} w_j (\Delta c)^j , \qquad \textbf{(2.9)}$$

where $\Delta \hat{p}$ is an estimate of Δp, and w_j (j = 1, 2, 3) are the coefficients to be determined using the neural network.

After the training of the network was completed, the measurement device was tested over a range of 0-20*MPa*; the results of the testing are shown in Figure 2.16. The preliminary results of the tests completed showed that the non-intrusive method was

accurate within one percent and decreased the time and effort compared to using direct pressure measurement devices.

Image has been removed temporarily due to Copyright

Figure 2.16: Results of FLNN validation for non-intrusive pressure measurement [Yu, 2005].

The work done by Yu showed that non-intrusive methods for measurement can be achieved using neural networks. Pollmeier et al [2004] conducted experiments at the University of Bath in the United Kingdom which used intrusive measurement devices along with neural networks to identify faults in fluid power systems. Tests were completed for both simulated and experimental systems; however, only the simulation results will be outlined in the following section because both simulation and experimental results were similar.

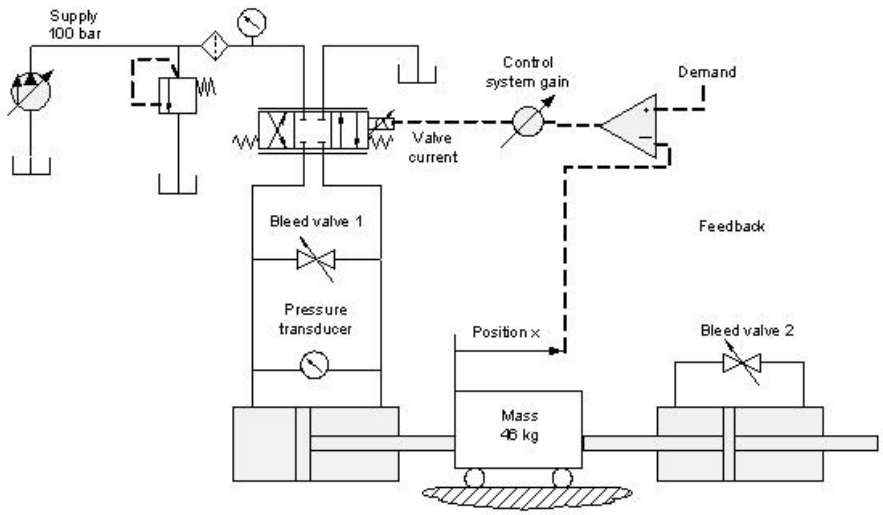


Figure 2.17: Schematic of the servo-valve controlled linear actuator system [Pollmeier, 2004].

The system being considered was a servo-valve controlled linear actuator system shown in Figure 2.17. Five types of faults were considered in the system; (i) increased leakage across the actuator piston, (ii) a change in load, (iii) a change in the mass moved by the actuator, (iv) a reduction in system supply pressure, (v) and an increase in friction

between the mass and the ground. Each of the five faults was given a fault range, and five fault levels were created which are outlined in Table 2.1.

Table 2.1: Parameter fault levels for simulation study

Fault Parameter	Fault Level 1	Fault Level 2	Fault Level 3	Fault Level 4	Fault Level 5	Parameter Unit
Leakage	0.0	0.1	0.2	0.3	0.4	*L/min/bar*
Load	0	500	1000	1500	2000	*N*
Mass	50	100	150	200	250	*kg*
Pressure	100	80	60	40	20	*bar*
Friction	1000	8000	15000	22000	29000	*N/m/s*

The first type of diagnostic system implemented was the use of five networks, one for each fault type, with each network containing thirty-five input neurons, one hidden layer containing fifteen neurons, and a single neuron representing the fault considered. For the input neurons all five fault parameters were used as inputs with each input receiving six additional delayed time steps. Their results showed that the classification using five separate networks with each diagnosing a specific fault was accurate except for the mass which showed significant scatter. It was concluded by the authors that the scatter was caused by an insufficient excitation of the mass in the duty cycle.

The results of diagnosing the faults using five separate networks were accurate but it was necessary to run each of the networks separately to obtain a proper fault. To overcome this, a secondary network was created which would allow the five single fault networks to run in parallel and feed into another network placed in series which could differentiate between the signals to obtain the appropriate fault. The second network was a 25:10:5 feed-forward network; each of the single networks input five time steps of data and each output corresponds to a specific fault. The results of the fault detection network trained with a secondary network showed little scatter in comparison to the single network results because in an effort to simplify the inputs to the secondary network, the outputs of the primary networks were given a specific value depending on what range the output lied in.

Although the results show that the trained network worked well at identifying a single fault, there were still discrepancies on the proper fault levels for certain cases. Each fault type was trained with five-hundred data points with each fault level receiving one-hundred points; this means that the first fault level, which was the no fault situation, received five times more training points than any other level. To alleviate the dependence of training on one case, two methods may be undertaken; decrease the number of first fault level training sets so the case is not as dominant. Another possibility is increasing the learning rate (which will be discussed in Chapter 3), enabling the other fault levels to increase their dominance.

2.5.3 Modeling of Fluid Power Systems Using Neural Networks

As mentioned previously, fluid power systems are highly non-linear and operate over a large range. Similarly to controlling fluid power systems, modeling such systems can become difficult using traditional methods. The difficulty encountered lies in the dependence on linear equations to describe fluid power systems; these equations can be made by linearizing the system at specific operating conditions but as the operating conditions change the ability to accurately describe the system deteriorates.

One solution to overcome this issue is using neural networks to model hydraulic systems. The Fluid Power Centre at the University of Bath has conducted extensive research in the use neural networks for modeling fluid power systems. In a study conducted by McNamara et al [1997] the use of a hybrid model composed of both analytical and neural network components was considered for a variable displacement piston pump shown in Figure 2.18. The neural network component of the hybrid model was used to model the pump dynamics, and the servomechanism was modeled using Bath*fp*; a program created at the University of Bath specifically for fluid power applications.

Figure 2.18: Swash-plate piston pump configuration [McNamara, 1997].

To model the pump dynamics, a neural network was created using delivery pressure, swash-plate angle and velocity as inputs, and the swash-plate operating moment as the output. The network chosen was a feed-forward using backpropogation to train the network. Three input neurons were used which corresponded to the inputs listed above. Twelve neurons were used in the hidden layer, and one neuron was used in the output layer for the swash-plate operating moment. The network was trained using data which covered the entire operating range of the pump and a mean square error of 0.03 was achieved. A comparison of the actual operating moment and the neural network operating moment is given in Figure 2.19.

Figure 2.19: Mean operating moment calculated using neural network validation data [McNamara, 1997].

Once the neural network was trained to model the torque characteristics properly, a hybrid model was created. The hybrid model was tested using a variety of pressures at varying demand scenarios. The results for a step increase followed by a step decrease at 200*bar* and 50*bar* are shown in Figure 2.20. It can be seen that for lower pressures the negligence of velocity dependence which is a component of linearized models is acceptable; but as the pressure increases this dependency leads to a substantial error.

Figure 2.20: Hybrid model simulation results for a step up-step down demand cycle [McNamara, 1997].

2.6 Summary

Neural networks have been shown to be a viable solution to overcome the dynamic and non-linear characteristics for controlling, monitoring and modeling of fluid power systems. The networks have been both static and dynamic in form. The following chapters will discuss the training of the neural networks with a focus on modeling dynamic systems.

Chapter 3: Training of Dynamic Neural Networks

3.1 Introduction

A variety of neural network structure types were previously discussed in Chapter 2, and although the type of network chosen for a specific task is important, an equally important consideration is the type of training algorithm used. Training a neural network is done to minimize the error of a neural network output for the training data entered as described in equations 3.1 and 3.2. The training of the network will progress until criteria are met which are set by the user; examples of these criteria are error minimization or the number of times an update of the neural weights has occurred.

The error of a neural network is the difference between the desired output and the actual output produced by the network,

$$e(k) = y_d(k) - y(k). \tag{3.1}$$

In the above equation the error (e), desired output (y_d) and network output (y) are all given at the k^{th} time step. Although there are a variety of error criteria which can be used [Gupta, 2003][Haykin, 1999][Goulermas, 2007], the most common error criteria is the minimization of the mean squared error; the mean squared error will be the basis for error criterion in the following discussions.

For a given time step k the least squared error is given by,

$$E(k) = \frac{1}{2}e(k)^2. \tag{3.2}$$

Once the error criterion is been established consideration must be paid to when the training procedure will occur. Training adaptations for neural networks can occur in two main forms; continuous training, commonly referred to as instantaneous training, or epochwise training, also known as batch training. For instantaneous training the weights are updated as the training set is run through the network; after each time step the error is

calculated using Equation (3.2) and weights are changed based on the algorithm chosen which will be discussed in detail later.

Batch training does not adjust the weights after each time step, but the error is calculated at each time step and then the average error is calculated for an entire training set. The mean squared error for batch training is,

$$E = \frac{1}{2}\frac{1}{N}\sum_{m=1}^{N} e(m)^2 \ , \qquad \textbf{(3.3)}$$

where m is the time step being considered and N is the number of time steps in a training set. This means that for a network which describes a dynamic system (for example, if a small steady state error is found but a large dynamic error occurs), then the network is trained not just using information from one state, but from both states.

This chapter will explore a variety of types of training algorithms used in the training of neural networks. Neural network training methods are broken down into two main categories; gradient and non-gradient training methods. This chapter is used to explore the fundamentals involved for both categories of training.

3.2 Gradient Training Methods

In this section, the background behind gradient training methods is described. Although only non-gradient methods are used for training networks in the research performed, to show the intricate nature of gradient training and their pitfalls, gradient networks will be explained thoroughly. Please refer to the references cited for greater detail. The use of gradient training methods is very common in neural networks due to their ability to find answers for even the most complicated of systems [Gupta, 2003][Haykin, 1999]. The gradient training method works on the concept of gradient descent; the gradient of the error is calculated and weights are changed based on the slope of the error gradient.

The most common form of the gradient based training method is backpropagation (BP). BP uses the gradient of the error to create a signal which is sent back through the neuron to correct the synaptic weights. The gradient estimate of the weighting vector using mean squared error is given by,

$$\nabla W\left(e^2(k)\right) = \frac{\partial e^2(k)}{\partial W(k)} = 2e(k)\frac{\partial e(k)}{\partial W(k)}. \tag{3.4}$$

Consider a schematic representation of BP for a single neuron shown in Figure 3.1. Given that for the sigmoid element shown which contains the neuron output *s(k)*, and the activation function *σ(s(k))*,

$$\begin{aligned} e(k) &= y_d(k) - y(k) \\ e(k) &= y_d(k) - \sigma(s(k)) \\ e(k) &= y_d(k) - \sigma\left(W^T X(k)\right) \end{aligned} \quad . \tag{3.5}$$

If the partial differential is taken of Equation (3.4) with respect to the indicated synaptic weight, it can be shown that,

$$\nabla W\left(e^2(k)\right) = -2\mu e(k)\sigma'(s(k))X(k). \tag{3.6}$$

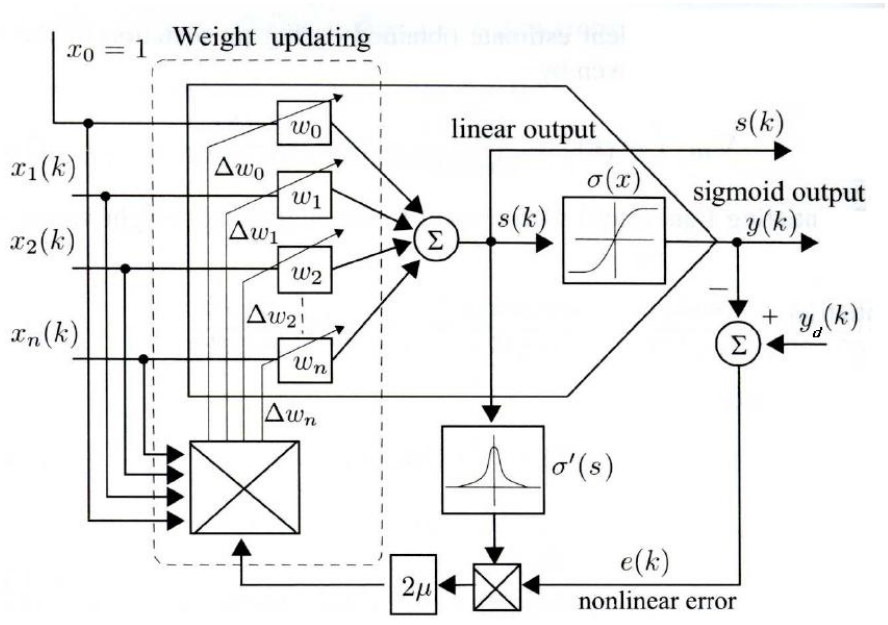


Figure 3.1: Schematic representation of BP for a single neuron.

Figure 3.1 shows schematically the BP process for a single neuron. In the above equation μ is the learning rate; the learning rate affects the speed of the minimization procedure as well as the algorithms stability. If μ is large then the step size of the gradient is large and the optimum point will be reached quickly. However if μ is too large then the step to the next point will overshoot the minimum error; this process will repeat itself and oscillate over the minimum error point causing the training process to become unstable. At the same time if μ is chosen to be too small then very small steps are taken which can cause a very slow training process.

Because a μ chosen to be too large or too small can cause deterioration in BP abilities, an appropriate μ must be chosen which causes a problem because the optimum μ is system dependent. This can be solved through trial and error, however a good range for μ is from 0.1 to 1 [Gupta, 2003]. Given Equation (3.6) the updating algorithm for the augmented weight vector for a single neuron is given by,

$$\begin{aligned} W(k+1) &= W(k) - \nabla W(k) \\ W(k+1) &= W(k) + 2\mu e(k)\sigma'(s(k))X(k) \end{aligned}. \qquad \textbf{(3.7)}$$

The above updating mechanism works well for a single neuron or a system with only one input layer and one output layer, but when a multilayer feed-forward neural network (MFFNN) is used BP becomes more complicated. The main complication is derived from the calculation of error for the neuron. In Equations (3.1) and (3.2) the error is calculated using the desired output, y_d, as a comparison. For MFFNNs the output comparison for each neuron becomes a problem as only the final desired output is known.

To overcome the shortcomings of BP the partial derivatives for error, $\delta(k)$, are considered. For a given neuron i located in the j^{th} layer the partial derivative for error with respect to the neuron output is,

$$\delta_i^{(j)}(k) \stackrel{\Delta}{=} -\frac{1}{2}\frac{\partial e^2(k)}{\partial s_i^{(j)}(k)}. \qquad \textbf{(3.8)}$$

If the network contains one output neuron with M layers, and $j = M$, then,

$$\begin{aligned}\delta_1^{(M)}(k) &= e_1^{(M)}(k)\sigma'\left(s_1^{(M)}(k)\right) \\ \delta_1^{(M)}(k) &= e(k)\sigma'\left(s_1^{(M)}(k)\right)\end{aligned} \quad . \tag{3.9}$$

If $j \neq M$ then the chain rule must be applied,

$$\begin{aligned}\delta_i^{(j)}(k) &= -\frac{1}{2}\frac{\partial e^2(k)}{\partial s_i^{(j)}(k)} = -\frac{1}{2}\frac{\partial e^2(k)}{\partial s_1^{(M)}(k)}\frac{\partial s_1^{(M)}(k)}{\partial s_i^{(j)}(k)} \\ \delta_i^{(j)}(k) &= \delta_1^{(M)}(k)\frac{\partial s_1^{(M)}(k)}{\partial s_i^{(j)}(k)}\end{aligned} \quad . \tag{3.10}$$

Using Equation 2.1, the linear combiner component of the synaptic operation may be used to solve the partial derivative in Equation (3.10). For layer j, where $j = M - 1$,

$$\begin{aligned}\frac{\partial s_1^{(M)}(k)}{\partial s_i^{(j)}(k)} &= \frac{\partial\left(W_1^{(M)^T}(k)\overline{y^j}(k)\right)}{\partial s_i^{(j)}} = \frac{\partial\left(W_1^{(M)^T}(k)\sigma\left(\overline{s^j}(k)\right)\right)}{\partial s_i^{(j)}} \\ \frac{\partial s_1^{(M)}(k)}{\partial s_i^{(j)}(k)} &= w_{1i}^{(M)}(k)\sigma'\left(s_i^{(j)}(k)\right)\end{aligned} \quad . \tag{3.11}$$

Substituting Equation (3.11) into (3.10) yields,

$$\delta_i^{(j)}(k) = \delta_1^{(M)}(k)w_{1i}^{(M)}(k)\sigma'\left(s_i^{(j)}(k)\right). \tag{3.12}$$

In the above equation all of the information may be calculated based on either information from the neuron being changed or the output of the network. If the layer being considered is not the layer before the output layer the methodology remains the same; the chain rule can be used to expand the layer under consideration back to the output layer using the intermediate layers. Using a linearized approximation, it is possible to obtain the error gradient for a specific neuron in any layer based on the output information.

To update the weights the methodology remains similar to Equation (3.7), however now the gradient of the weighting matrix changes as,

$$\nabla W_i^{(j)}\left(e^2(k)\right)=\frac{\partial e^2(k)}{\partial W_i^{(j)}(k)}=\frac{\partial e^2(k)}{\partial s_i^j(k)}\frac{\partial s_i^j(k)}{\partial W_i^{(j)}(k)}=2\left(-\frac{1}{2}\frac{\partial e^2(k)}{\partial s_i^j(k)}\right)\frac{\partial s_i^j(k)}{\partial W_i^{(j)}(k)}$$
$$\nabla W_i^{(j)}\left(e^2(k)\right)=-2\delta_i^j\frac{\partial s_i^j(k)}{\partial W_i^{(j)}(k)} \quad \textbf{(3.13)}$$

From the synaptic output of the linear combiner,

$$s_i^j(k)=\left(W_i^j(k)\right)^T\bar{y}^{(j-1)}$$
$$\frac{\partial s_i^j(k)}{\partial W_i^{(j)}(k)}=\bar{y}^{(j-1)} \quad \textbf{(3.14)}$$

Therefore equation (3.13) becomes,

$$\nabla W_i^{(j)}\left(e^2(k)\right)=-2\delta_i^{(j)}\bar{y}^{(j-1)}. \quad \textbf{(3.15)}$$

Substituting Equations (3.9), (3.12) and (3.15) into Equation (3.7) yields the weight adaptation criteria for any neuron in any layer of a MFFNN,

$$W(k+1)=W(k)-\nabla W(k)$$
$$W(k+1)=W(k)+2\mu e_i^{(j)}(k)\sigma'\left(s_i^{(j)}(k)\right)\bar{y}^{(j-1)}. \quad \textbf{(3.16)}$$

For a MFFNN the algorithm for BP can be derived even for complicated morphologies; however, it should be noted that MFFNNs are static networks. Applying BP to a dynamic network requires more consideration. One possibility is to treat the dynamic connections as static connections and change the weights as previously elaborated upon [Werbos, 1990][Gupta, 2003][Li, 2007]. Another method to use BP for dynamic networks is to *"unfold"* the dynamic components of a network and train the network as a static network whose morphology changes at each time step [Haykin, 2003]. However when δ must be found for time delayed errors the training structure can become very difficult to solve.

BP training methods have been shown to be very useful and have a wide variety of adaptations for both static and dynamic networks [Haykin, 1999][Gupta, 2003]. As mentioned above when BP is applied to dynamic networks there is one drawback to the dynamic component; the algorithms ability to properly handle the dynamic structure.

The homogeneous treatment of the network can be overcome but this can still dramatically affect the robustness of the training.

Another major flaw when using BP is the inability to move from a local minima to a global minima [Gupta, 2003][Andersson, 2001][Goldberg,1989]. Figure 3.2 shows a simple example of global minimum training issues with BP. In order to properly train a BP network not only must a proper learning rate be picked, but the initial setting of weighting parameters must be within the global minimum slope. As seen in Figure 3.2 if the initial weights are picked on interval *b* then the global minimum will be obtained. However if weights are picked on interval *a*, one of the local minima will be found with gradient training. To obtain an appropriate initial weight set either the user must have prior information of the system being considered, or many attempts must be made to ensure the minimum has been reached. The former eliminates the black box concept often desired in neural networks and the latter can increase training time while at the same time not guaranteeing a minimum error.

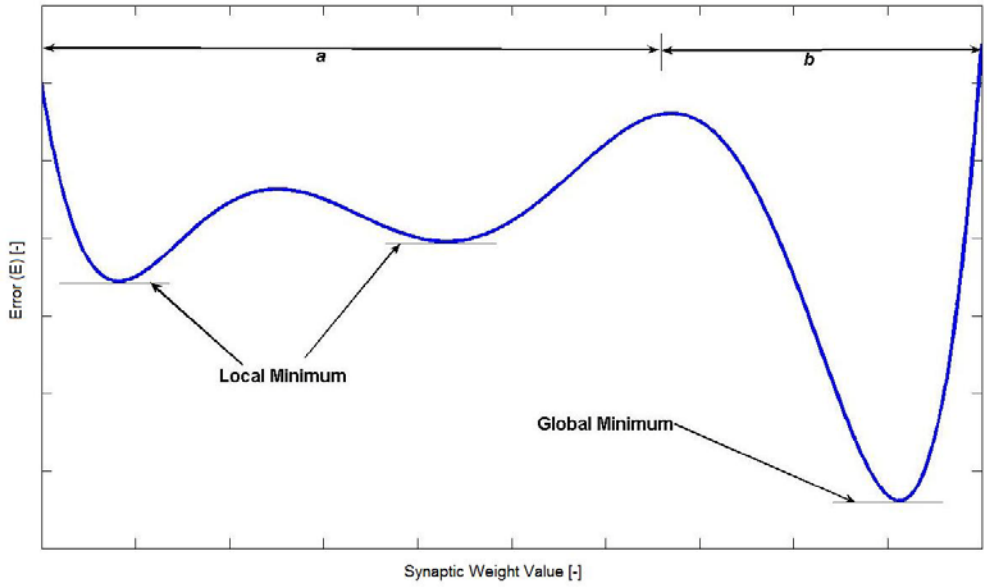


Figure 3.2: A one-dimensional example of a global minimum and local minimum error search.

3.3 Non-Gradient Training Methods

As the application of neural networks have become increasingly complicated and the need for more robust training methods have become more desirable, the types of optimization algorithms being applied to neural network training has also changed. The use of BP has been explored, but because of the drawbacks stated above the use of non-gradient optimization methods has increased. In the following section the use of non-gradient methods will be discussed, with an emphasis being placed on evolutionary based algorithms. Although the main focus of training of this thesis is the training of RGNNs using the complex algorithm, the genetic algorithm is presented first because it is a more traditional non-gradient training method.

3.3.1 Genetic Algorithm for Neural Network Training

Like many evolutionary optimization methods, genetic algorithms (GA) are derived from biological systems in nature much like neural networks [Haykin, 2003]. The concept of the GA is simple; in a population there are certain "individuals" (a set of neural weights) who are better fit for a specific task than other individuals, much like the infamous Darwinian theory regarding "survival of the fittest" and "natural selection" [Darwin, 1859]. Once the proper individuals have been obtained they spawn new individuals which are similar but provide a better fit to the problem at hand (a set of optimal weight values that minimize the error, for example).

Two major advantages of genetic algorithms are that they are relatively unconstrained by limitations such as continuity and the existence of the training function derivative [Goldberg, 1989]. The second advantage is particularly important in neural networks. As stated in the previous section, BP is dependent on the derivative to acquire the gradient which is used to minimize the output error of the network. In alleviating the need for a gradient, using GA also eliminates the need to find the derivative of the activation function.

GAs are composed of three main operators; reproduction, crossover and mutation all of which will be defined in the next section. As stated above, genetic algorithms are based on the concept of survival of the fittest. To facilitate discussion, the neural network shown in Figure 3.3 will be used to assist in defining the steps involved in GA training.

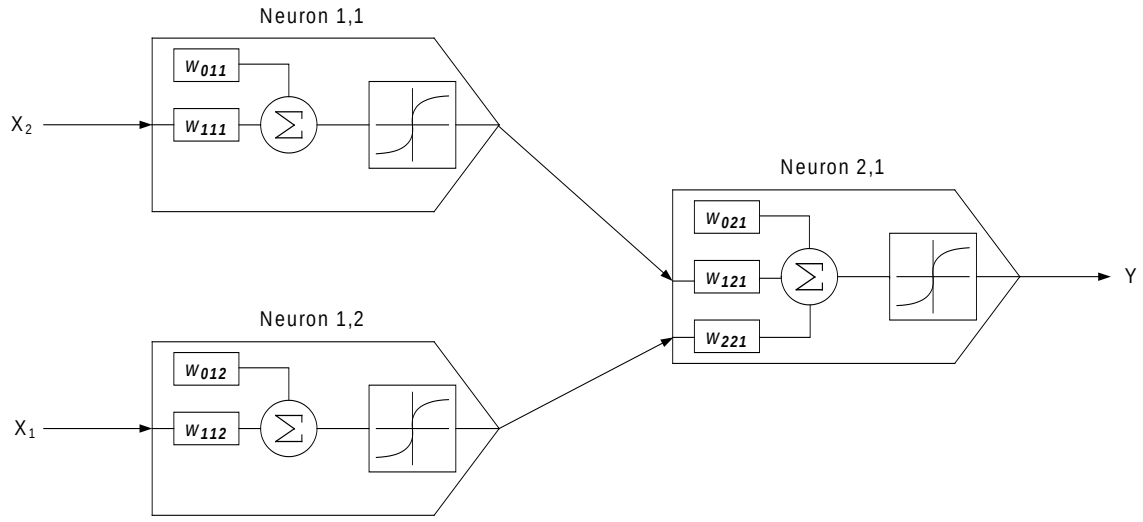


Figure 3.3: Schematic of example static network consisting of two inputs and one output.

The neural network in Figure 3.3 contains four synaptic weight connections (w_{111}, w_{112}, w_{121}, w_{221}), and three bias weights (w_{011}, w_{012}, w_{021}). The weight matrix is written as,

$$W = \begin{bmatrix} w_{011} & w_{111} & w_{012} & w_{112} & w_{021} & w_{121} & w_{221} \end{bmatrix}. \quad \textbf{(3.17)}$$

The function that the neural network will be trained to "mimic" is a SISO system described by the mathematical function,

$$y = x + x^{1/2} + 2. \quad \textbf{(3.18)}$$

The first step in training a network using GAs is to create an "initial population"; the initial population can have a variety of initiation conditions set by the user, but one property which must be adhered to is a population whose size is an even numbered. The importance of this property will be discussed later. A population is comprised of a number of "individuals", with each individual corresponding to a set of weights for the

neural network (the weights shown in Equation 3.17). The minimum number of individuals in a population is the number of weights being solved for plus one. For the example given in Figure 3.3, there are seven weights present, this means that the minimum number of individuals needed is eight. For the example considered, an initial population range of [0, 1] was considered. Table 3.1 shows the initial population for the network considered (weights are randomly chosen). Once an initial population has been formed, the next step is to calculate the output error for each individual, $E^{(i)}$, using Equation 3.3. Repeated for clarity, $E = \frac{1}{2}\frac{1}{N}\sum_{m=1}^{N} e(m)^2$.

The procedure is to assign individual 1 with 7 random weights as illustrated in Table 3.1. The individual is then given an input with a size N; the same signal is input into equation 3.18. The outputs of the NN and the equations are compared and the error calculated and using Equation 3.3, the output error for each individual, $E^{(i)}$, is determined. This procedure is repeated for each individual.

Table 3.1: Initial population with corresponding error for FFNN. Note: individual 2 has the lowest error.

Individual	w_{011}	w_{111}	w_{012}	w_{112}	w_{021}	w_{121}	w_{221}	$E^{(i)}$
1	0.8382	0.8292	0.7142	0.7912	0.4799	0.0934	0.9917	45440.20
2	0.3902	0.4406	0.0066	0.7777	0.9006	0.7098	0.5805	42185.88
3	0.9181	0.7549	0.5861	0.8833	0.0265	0.4554	0.9461	46186.07
4	0.1559	0.9671	0.5759	0.4114	0.1029	0.7538	0.2423	48011.48
5	0.1680	0.7507	0.9734	0.2773	0.3415	0.4124	0.7218	45950.36
6	0.4612	0.4823	0.9062	0.0180	0.9137	0.8917	0.0713	43871.46
7	0.1093	0.0980	0.7658	0.9466	0.5638	0.4087	0.5459	46070.98
8	0.7010	0.2208	0.0387	0.8139	0.7986	0.4011	0.4596	44972.19
							E_p	362688.6

Before any updates can be done to the population, an intermediate (or tentative) population must be formed using the "survival of the fittest" criteria. The tentative population is not a new population (each individual's weights are the same but the placement of their numerical value in the matrix shown in Table 3.1 re-arranged); it is

just one step towards obtaining a new population. To choose a tentative population the error of each individual is used to evaluate the individuals "fitness factor". The fitness factor helps described how robust each individual is for the input data; the fitness factor, $f^{(i)}$, is calculated by,

$$f^{(i)} = \frac{E_p - E^{(i)}}{E_p}. \qquad \textbf{(3.19)}$$

Equation 3.19 shows that the lower the individual error $E^{(i)}$, the more fit the individual will be [Song, 1998]. The fitness factor for each individual and the population's total fitness is given in Table 3.2.

Table 3.2: Values for individual fitness factors and total fitness of initial population.
Note: individual 2 has the best fitness.

Individual	1	2	3	4	5	6	7	8	total
$f^{(i)}$	0.8747	0.8837	0.8727	0.8676	0.8733	0.8790	0.8730	0.8760	7.0000

The next step in creating a tentative population is creating a probability wheel [Goldberg, 1989]. Figure 3.4 shows a probability wheel example. The basis of a probability wheel function is as follows, the more fit an individual is, the more space that individual takes up on the wheel. When the wheel is "spun" the chance of the wheel landing on a specific individual increases as the fitness of that individual increases. As there is no wheel function in computer coding, the probability wheel was altered to create a probability matrix.

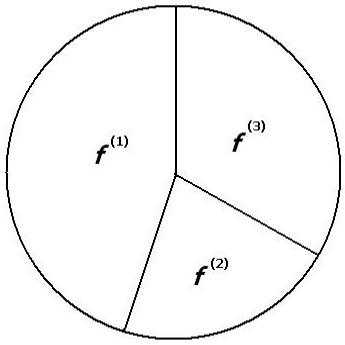


Figure 3.4: Physical representation of reproduction probability for GA training [Goldberg, 1989].

The fitness factor of each individual was used to create a fitness ratio,

$$f_{ratio}^{(i)} = \frac{f^{(1)}}{f_p}. \tag{3.20}$$

where f_p is the fitness of the population found by summing the fitness for all the individuals in the population. Table 3.3 shows the fitness ratios for each individual for the initial weights. When all of the fitness ratios for a population are added together, the summation is equal to one. Therefore the fitness ration corresponds to how much of the probability circle each individual occupies if the circumference of the circle is one. However, as stated before, computer programming methods do not contain a spinning wheel. The approach used here was to create a fitness range for each individual instead of an actual wheel.

Table 3.3: Values for individual fitness factors and total fitness of initial population.
Note: as would be expected, individual 2 has the highest fitness factor.

Individual	1	2	3	4	5	6	7	8	total
$f_{ratio}^{(i)}$	0.1250	0.1262	0.1247	0.1239	0.1248	0.1256	0.1247	0.1251	1.0000

The fitness ratios are used to create a "fitness range" table as shown in Table 3.4. The first individual has a range of zero to its fitness ratio. The second and subsequent individuals have a fitness range of,

$$\begin{aligned} f_{range,\min}^{(i)} &= f_{range,\max}^{(i-1)} \\ f_{range,\max}^{(i)} &= f_{range,\min}^{(i)} + f_{ratio}^{(i)} \end{aligned}. \tag{3.21}$$

Table 3.4: Fitness range of each individual in initial population.

Individual	1	2	3	4	5	6	7	8
$f_{range,min}^{(i)}$	0.0000	0.1250	0.2512	0.3759	0.4998	0.6246	0.7501	0.8749
$f_{range,max}^{(i)}$	0.1250	0.2512	0.3759	0.4998	0.6246	0.7501	0.8749	1.0000

The next step is to create a matrix which is the same size as the initial population. The numbers in the matrix are comprised of random numbers between zero and one

(defined as *Rand* in this work). The purpose of the matrix is to pick a tentative new population based on the random values and the fitness range. Take for example the partial random population matrix,

$$Rand = \begin{bmatrix} 0.1145 & 0.7861 & . & . & . \\ . & . & . & & \\ . & . & & . & \\ . & . & & & . \end{bmatrix}.$$

Rand's column size is equal to the number of weights in the network, and *Rand*'s row size is the number of individuals in the population. It can be seen using Table 3.4 that the value in the first row and column of the random matrix falls in the fitness range of individual 1, and the second column first row random value falls into the range of individual 7. The same analysis is done for each cell of the random matrix, and this is used to form an index matrix,

$$Index = \begin{bmatrix} 1 & 7 & . & . & . \\ . & . & . & & \\ . & . & & . & \\ . & . & & & . \end{bmatrix},$$

where each cell in the matrix corresponds to the individual whose weight will fill the tentative new matrix. It is important to note that the "1" corresponds to the first weight in individual 1, (because its location is now the first row and first column of this new matrix) the 7 corresponds to the second weight in the individual seven (because its location is the first row and second column of the new matrix). The same methodology is performed for the entire index matrix. So for instance, using Table 3.1 as a reference, the tentative new matrix represented by W_{tent} is,

$$W_{tent} = \begin{bmatrix} 0.8382 & 0.0980 & . & . & . \\ . & . & . & & \\ . & . & & . & \\ . & . & & & . \end{bmatrix}.$$

Again, it must be emphasized that only the placement of the weights in the new matrix has been changed with this step. Thus, the tentative individual 1 now has weights [0.8382 0.0980 etc.].

Once a tentative population is formed, the calculation of a new population can begin. The first step in the calculation of the new population is taking each individual in the tentative population and matching it with another individual to "mate" with. The process of mating (to be discussed below) is what creates the new population based on the characteristics of the tentative population (some of the weights are now actually changed). Mates are randomly chosen for each individual; because each individual needs a mate it is imperative that an even number of individuals are chosen for the initial population. For the example being considered, suppose individuals one and five were randomly chosen to be paired together. Before mating, the weights of the tentative individuals are,

$$W_{tent} = \begin{bmatrix} 0.8382 & . & . & . \\ . & & & \\ . & & & \\ 0.1559 & . & . & . \\ . & & & \\ . & & & \end{bmatrix}.$$

The process of coming up with new individuals from mating is achieved by picking a mate for each individual. The next step of mating is to perform "crossover"; crossover involves creating new individuals based on the tentative individuals. Standard GAs use binary values only for the weights [Goldberg, 1989], so crossover becomes simple by just splitting the binary segment and crossing values (switching some of the binary numbers) between individuals. However, for floating point weight systems this method cannot be used. The method considered is subsequent sections is defined as a "floating-point" crossover method [Mahanti, 2005]. Unlike standard GAs which creates two new individuals, or "children", which are not identical to the tentative individuals, or "parents", heuristic crossover will create one child which is comprised of both parents genetic material (weights defined by Equation 3.22, and another child which is a "clone" of the parent with the best fitness (all the weights are the same as shown in Equation 3.23). The new individuals (children) using heuristic crossover are calculated by,

$$I^{B(new)} = I^{A} + r\left(I^{A} - I^{B}\right), \quad \textbf{(3.22)}$$

$$I^{A(new)} = I^{A} \quad \textbf{(3.23)}$$

where I^A is the tentative individual with the smallest error, I^B is the tentative individual with the largest error, and r is a random value between zero and one.

For the tentative individuals who were mated (1 and 5), the error for tentative individual 1 is 44002 and tentative individual five has an error of 44010 (an intermediate step not shown in Table 3.2). Therefore, for w_{011} with $r = 0.5$,

$$w_{011}{}^{5(new)} = w_{011}{}^{1} + r\left(w_{011}{}^{1} - w_{011}{}^{5}\right) = 0.8382 + 0.5\left(0.8382 - 0.1559\right)$$

$$w_{011}{}^{5(new)} = 1.1794,$$

for one of the new individuals and,

$$w_{011}{}^{1(new)} = w_{011}{}^{1} = 0.8382$$

for the second individual (clone). This process is repeated for all mating pairs in the population. A new matrix of individuals now exists in which half of the individuals are unchanged (clones) and the rest are changed using the aforementioned steps.

The final step in the mating process is "mutation". Mutation is the process of randomly adjusting weights away from the genetic trend; this can occur in a wide variety of patterns ranging from one weight in every individual being changed at every generation to one weight in the entire population being changed at each generation. The purpose of mutation is to create individuals which stray away from the genetic path and attempt to find other optimization points. Therefore, if a population is being saturated with poor genetics, then mutation will attempt to set the population on another path.

The type of mutation being considered for the following research is non-uniform mutation. Non-uniform mutation is governed by,

$$w^{i(mutate)} = w^{i} + \left(b - w^{i}\right) f\left(G\right) \qquad \text{If } r_1 < 0.5 \quad \textbf{(3.24)}$$

$$w^{i(mutate)} = w^{i} - \left(a + w^{i}\right) f\left(G\right) \qquad \text{If } r_1 \geq 0.5 \quad \textbf{(3.25)}$$

where,

$$f(G) = \left(r_2 \left(1 - \frac{G}{G_{\max}} \right) \right)^c . \quad \textbf{(3.26)}$$

In Equations 3.24 and 3.25, r_1 and r_2 are random numbers between zero and one, b and a are the upper and lower boundaries respectively of the possible weight range given by the user. G is the number of times a new population has been calculated (this is also referred to as the number of generations). G_{max} is the maximum number of generations, and c is a system parameter determining the degree of non-uniformity.

To illustrate mutation consider w_{011} in the new population with a boundary of [-5, 5], a non-uniformity parameter of c = 3, and r_1 and r_2 equal 0.3 and 0.6 respectively. If the maximum number of generations is G_{max} = 50 and the current generation is G = 10 then,

$$f(G) = \left(0.6 \left(1 - \frac{10}{50} \right) \right)^3 = 0.1106 ,$$

and since $r_1 < 0.5$,

$$w_{011}^{(mutate)} = 0.8382 + (5 - 0.8382)(0.1106)$$
$$w_{011}^{(mutate)} = 1.2985 .$$

Once a new population is created the criteria are tested for and the process repeats itself until the training criteria are met as shown in Figure 3.5. Although large populations may be needed to create an independent set of individuals, the genetic algorithm can prove very useful in networks with multiple optimization points. Because of the reproduction and crossover components of GAs the chances of becoming caught in a local minimum are much smaller than BP, which is a major advantage.

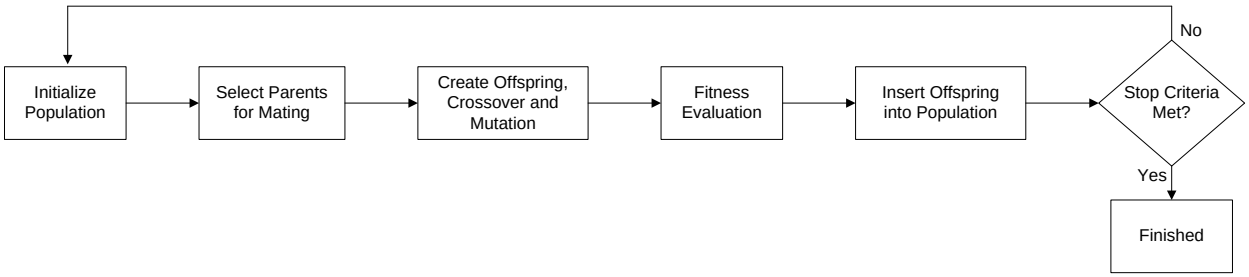


Figure 3.5: Flow diagram of genetic algorithm [Andersson, 2001].

In addition to an increased chance of finding the global minimum, GAs also contain a failsafe to ensure that once an optimization point is found it is in fact the global optimization point. This is done with the mutation operation which allows weights to be changed randomly and be set off the general course chosen by the rest of the population. If a mutation obtains an individual which is fit then the entire population will begin to drift towards the individual in subsequent generations.

3.2.2. Complex Algorithm for Neural Network Training

As mentioned previously, the use of non-gradient network training has become more viable as the complexity of neural networks has increased to include such things as dynamics. The complex algorithm (CA) optimization method is an evolutionary algorithm which works similar in nature to GA. Just like GA, CA uses a population of points referred to as individuals which for neural networks are synaptic weights. Unlike GA, CA does not modify the entire population in the creation of a new generation; instead at every generation one new point is made to replace the worst point in the previous generation [Andersson, 2001][Wiens, 2008a][Wiens, 2008b].

Just like when using GAs, to initiate the process for CAs first a population of individuals containing network weights must be produced that meet the criteria of the user. The minimum size of the population is $n \geq m+1$ [Andersson, 2001] where m is the number of variables being solved (for which for a neural network is the number of weights in an individual); in practice $n \geq 2m$. Next the fitness of each individual is obtained based on user criteria. The fitness factor considered for CA is minimization of

error which was also used for GA. Once the fitness of each individual is found, the individual with the worst fitness must be found in order for it to be replaced.

To illustrate the methodology behind the CA, the FFNN used for the GA example in the previous section will now be used to illustrate how the population is updated using the CA. The initial population under consideration is the same as Table 3.1; unlike GAs, CAs do not use a probability wheel based on a fitness function, instead they use the comparison of error directly. The initial population along with the corresponding error for each population is shown in Table 3.5.

Table 3.5: Initial population and error values for FFNN using CA.

Individual	w_{011}	w_{111}	w_{012}	w_{112}	w_{021}	w_{121}	w_{221}	E_i
1	0.8382	0.8292	0.7142	0.7912	0.4799	0.0934	0.9917	45440.20
2	0.3902	0.4406	0.0066	0.7777	0.9006	0.7098	0.5805	42185.88
3	0.9181	0.7549	0.5861	0.8833	0.0265	0.4554	0.9461	46186.07
4*	0.1559	0.9671	0.5759	0.4114	0.1029	0.7538	0.2423	48011.48
5	0.1680	0.7507	0.9734	0.2773	0.3415	0.4124	0.7218	45950.36
6	0.4612	0.4823	0.9062	0.0180	0.9137	0.8917	0.0713	43871.46
7	0.1093	0.0980	0.7658	0.9466	0.5638	0.4087	0.5459	46070.98
8	0.7010	0.2208	0.0387	0.8139	0.7986	0.4011	0.4596	44972.19

**denotes individual with highest error to be replaced.*

A summary of data for the following calculations is found in Table 3.6. The first step in CAs is to obtain a neural weight centroid, $\overline{W}$, which is found by calculating the average of all individuals excluding the individual with the highest error for each weight in the network,

$$\overline{W} = \frac{1}{n-1}\sum_{i=1}^{n} W^i \;, \quad where\, W^i \neq W^h. \tag{3.27}$$

The number of individuals in the population is n and W^h is the individual weighting matrix with the lowest fitness. A reflection individual must be created which is based on the placement of the centroid. The reflection individual, W^r, is,

$$W^r = \overline{W} + \alpha\left(\overline{W} - W^h\right). \tag{3.28}$$

The reflection coefficient, α, is used to give a reflection length about the centroid, $\overline{W}$. Andersson [2001] recommends a value of $\alpha = 1.3$.

One issue which can arise is the calculation of a reflection point having a lower fitness than its predecessor. The intent of CA is to manipulate the population one individual at a time while at the same time increasing the fitness of the worst individual. To overcome the conundrum of a lower fitness reflection point, a modified algorithm is suggested by Andersson [2001],

$$W^{r(new)} = \left[W^{r(old)} + \varepsilon\overline{W} + (1-\varepsilon)W^{l}\right]/2 + \left(\overline{W} - W^{l}\right)(1-\varepsilon)(2R-1), \qquad \textbf{(3.29)}$$

where,

$$\varepsilon = \left(\frac{n_r}{n_r + k_r - 1}\right)^{\frac{n_r + k_r - 1}{n^r}}. \qquad \textbf{(3.30)}$$

In Equations (3.22) and (3.23) R is a random number in the interval [0, 1], n_r is a constant which Andersson [2001] had chosen to be $n_r = 4$ (however it can be any integer), and k_r is the number of times the reflection individual has been repeated. A value for $W^{r(new)}$ is presented in Table 3.4; however, it should be noted that Equation 3.29 is only used if W^r produces a higher error than W^h. This process is repeated until the user criteria are met; a flow diagram of CA is given in Figure 3.6.

Table 3.6: Results for calculation of centroid and reflection points using CA.

	$\overline{W}$	W^r	$W^{r(new)}$
w^{011}	0.5123	0.9755	0.7439
w^{111}	0.5110	-0.0820	0.2145
w^{012}	0.5701	0.5627	0.5664
w^{112}	0.6440	0.9463	0.7952
w^{021}	0.5750	1.1886	0.8818
w^{121}	0.4818	0.1282	0.3050
w^{221}	0.6167	1.1034	0.8601
E_i		41693.37	42915.45

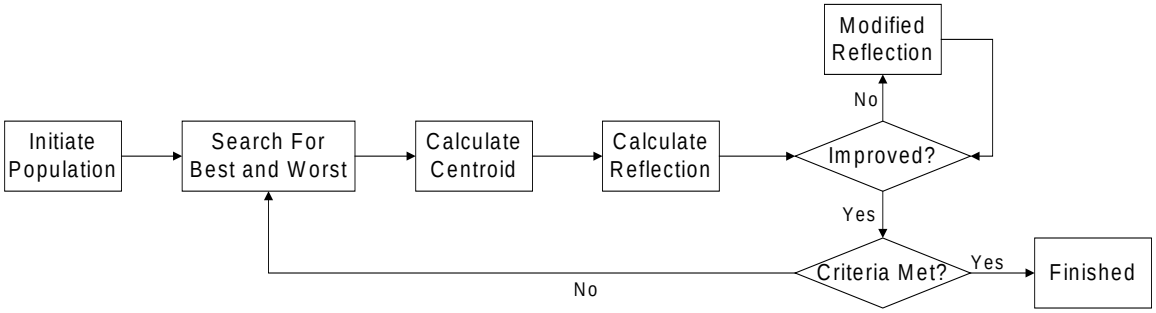


Figure 3.6: Flow diagram of the complex algorithm including modified algorithm.

As described for GA, one of the important applications of the mutation operator is maintaining the genetic material of the population. The modified reflection point attempts to accomplish the same feat but instead of random mutations, the mutation is based upon where it is and the number of times a new individual has been attempted for that centroid. The goal of the CA is to find the global minimum for error by changing the values of a specific synaptic weight connection in a population so the weight in each individual converges on a value which produces the global minimum; this is referred to as the "complex collapsing", or the population collapsing. With the addition of the modified reflection point the probability of the complex collapsing on the wrong point is reduced.

As mentioned in Chapter 2, CAs were used in tandem with RGNN to create a network which was used to train a simulation model for a load sensing pump by Wiens [2008b]. The combination of this network type and training method was shown to be effective and consideration of further studies using the pairing of RGNNS and CAs was suggested. In the following chapters the use of RGNN being trained with CA to create simulation models will be studied further in both the time and frequency domains. A comparison of results will be made with the application of GA to RGNN as a benchmark.

Chapter 4: Application of Complex Training Method to Recurrent Generalized Neural Network

4.1 Introduction

As discussed in Chapters 1 and 2, previous research had been completed at the U of S by Wiens [2008b] which used RGNNs in combination with CAs to create a model for a load sensing pump. The model was created by fine-tuning weights in the initial population until a set of weights was found which produced the desired results. As previously stated, the global objective of this thesis – and other past works – was to create a black box model for a load sensing pump using neural networks. Because Wiens' model was only achieved by fine tuning the weights, the model does not meet the global objective due to the fact that when the fine tuning of weights is conducted, the weight range used for training the system is known. When a neural network is found by using known information about the system – in this case, convergence criteria – it becomes a grey box model rather than a black box model.

However, because it was found that the RGNN and CA combination used by Wiens proved successful as a grey box model, the use of the pairing needed to be studied further. Initial studies by the author indicated that testing RGNNs using the CA even for simple systems could not produce an acceptable black box model. As a result, it was decided that the RGNN and CA combination needed further studies in order to explore their training capabilities. To do this the concept of applying a RGNN directly to load sensing pumps using experimental data was discarded in favour of using bench mark models from which definitive comparisons could be drawn. The bench mark so chosen was a basic linear-dynamic system.

The basic premise behind this approach was that if the RGNN could not be trained to a linear (known) simple model, then it would not be suitable for more complex nonlinear phenomena which exist in load sensing pumps. To test the "robustness" of

RGNNs when trained using the CA a "sensitivity" study was conducted. For the remainder of the literature, robustness refers to a networks ability to minimize training error, and sensitivity study refers to studying how sensitive the output response of a network was for the tests performed.

4.2 Selection of System for Creating Training Data

The following chapter will discuss the training of an RGNN using CAs to model a single input-single output (SISO) transfer function. The process was first to input a specified signal to both a SISO model and to the RGNN; then the output of both was compared and the error determined using the mean squared method error discussed in Chapter 3. Chapter 3 also discussed the use of batch training; batch training was employed for all studies to train RGNNs using the prescribed training method being tested. To ensure that the training algorithm for all cases did not become trapped in an infinite loop, three criteria were used to stop training; reaching a set number of iterations for the training mechanism, achieving an allowable maximum error, or reaching a set time limit. Although the maximum tolerable error was the same for all trained networks, the number of iterations and time allowed for training were set based on trial and error during initial training studies. It was found that if the number of iterations or time allowed were set to very large values, the probability of reaching an acceptable error did not increase.

An RGNN was trained with the use of two different input types; a three step input signal and a multiple sine wave input with three different frequencies, 0.5 *Hz*, 1 *Hz* and 5 *Hz*. Please note that here within; this sine based input will be referred to as a "frequency based input". The bench mark transfer function used as the basis for creating the necessary training data set was,

$$\frac{Y(s)}{X(s)} = \frac{s+4}{s^2+2s+9}. \qquad \textbf{(4.1)}$$

As discussed in Chapter 2, using discretization methods any transfer function can be represented using RGNNs. As shown in Appendix A, Equation 4.1 can be discretized by

linearizing the resulting derivatives obtained by taking the inverse Laplace transform which results in,

$$\frac{y^k - 2y^{k-1} + y^{k-2}}{\Delta t^2} + \frac{2y^k - 2y^{k-1}}{\Delta t} + 9y^k = \frac{x^k - x^{k-1}}{\Delta t} + 4x^k . \qquad \textbf{(4.2)}$$

Because the transfer function in known, an approximate RGNN equivalent output representation for Equation 4.2 is possible and is given in Figure 4.1. An in depth derivation for the connection weights is found in Appendix A. It must be again emphasized that if the training process cannot accommodate an exact form, then training a more generalized RGNN would be very difficult.

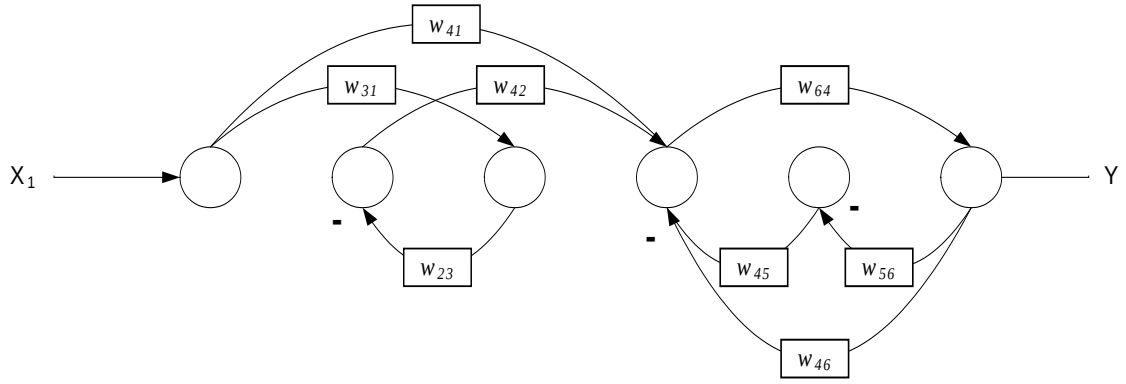


Figure 4.1: Approximate RGNN representation of transfer function to be modeled (found in Equation 4.1).

The first step in creating training data for both step response and frequency based inputs was to choose an appropriate time step. As stated in Chen [1994], for discrete (digital) control systems the sample rate must be at least ten times faster than the natural frequency of the system, or if frequency data is used, the fastest input frequency in order for the system to remain stable when the system is transformed from the continuous (s) domain to the discrete (z) domain. Since the RGNN is a discrete version of the system being studied, the natural frequency of the system in Equation 4.1 is,

$$\varpi_n = \sqrt{9\,rad^2/\sec^2} = 3\,rad/\sec = 0.955Hz \approx 1Hz .$$

Therefore, according to Chen, for step response the minimum sampled rate is 0.1 *seconds*. For the training of the RGNN using frequency information the smallest

frequency considered is 0.5 *Hz*; at a frequency of 0.5 *Hz* the minimum sample rate is 0.05 *seconds*. To help ensure stability based on the criteria established by Chen, a sample rate of 0.02 *seconds* was chosen for training the RGNN which meets the stability criteria for both step and frequency inputs.

It must be noted that even if the approximate weights for a RGNN are known, the discrete model does have error when compared to the output of the continuous model even at an acceptable sample rate. This error can create problems in the training of the RGNN and as such, the concept of establishing a bench mark to compare results was compromised. As a result, the output of the discretized model form of Equation 4.1 was used for training the RGNN instead of the output from Equation 4.1. Because the RGNN output was being compared to the discretized output, this problem was eliminated.

Using the output of the discretized model will create a new output which adheres to the stability criteria for digital systems posed by Chen. Using Equation 4.2, the output, y^k, and input, x^k, training data can be obtained for both step inputs and frequency based inputs. Then using the discretized transfer function in Equation 4.2 and the derivation found in Appendix A, an exact weight representation of the system under consideration using RGNNs with a sample time of 0.02 *seconds* is,

$$W = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0.0207 & -0.0192 & 0 & 0 & 0.9582 & -1.9547 \\ 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}. \qquad \textbf{(4.3)}$$

The above weight matrix is now used in the following sensitivity study.

4.3 Step Response Training of a RGNN Using CA

All networks which will be presented were trained with intent of achieving a maximum error in the population of less than 0.02, and this value is referred to as the

"maximum allowable error". It was found that while initially creating, training and testing RGNNS, a maximum allowable error value of 0.02 produced a network which did not contain any of the typical errors observed which will be discussed later. The value chosen for maximum allowable error was chosen arbitrarily based upon initial testing. When a weight or set of weights are said to be "optimized" then the set of weight – or weights – produce an output error which meets the maximum allowable error. One method used for the testing of RGNNs was to change the number of weights being optimized during the training process; the results of this method of testing will be discussed later in the thesis.

Four different weight formats were used to test the limitations of the number of weights being optimized; one weight, two weights, "only the required" weights and "all possible" weights. The training of two weights at a time is also referred to as weights being trained in "tandem"; this refers to weights trained simultaneously rather than one at a time. When a network is trained for only the required weights, it refers to all the non-zero weights from Equation 4.3 being optimized simultaneously. For the case where all possible weights are being trained, all possible weight connections and bias' described in Chapter 2 are being optimized (a truer "black box" situation).

The first type of training data considered was step response data. Step response data is commonly used as training data for neural networks because it contains both frequency data (at the changes in step), and magnitude information (the steady state information) which can be linear or non-linear depending on the system under consideration. As described in Chapter 1, a main advantage of neural networks is the implementation of black box theory. The size of the network was chosen to obtain the connections necessary to create an approximate representation of the transfer function being modeled. Although the size of the network chosen was considered to be sufficient for modeling the dynamics of the system under consideration, the first test was to view the ability of the network to obtain a RGNN model with no restrictions placed on which weight values were trained. Later on restrictions will be discussed which include the forming of initial populations and choosing to change only specific weights.

4.3.1 Training a RGNN Using Step Response with a Random Initial Population

After evaluating the values of the approximate weighting matrix in Equation 4.3 it was observed that the maximum weight value is the delayed feedback connection w_{46}, which is a value of approximately negative two. The weighting matrix also contains both positive and negative values, therefore as an initial test a RGNN was trained with an initial weight distribution of [-5, 5]. Figure 4.2 shows a typical response after training using random initial values for all weights including bias'.

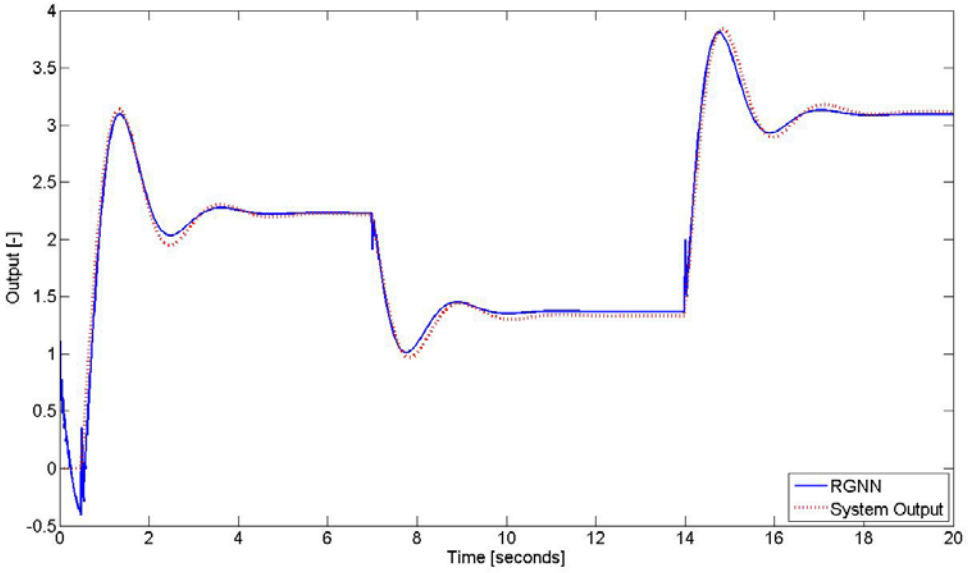


Figure 4.2: Output comparison of the desired response using step inputs to the output of a typical RGNN trained using a [-5, 5] initial distribution interval where all weights are trained.

There are three common error types which often occurred in the RGNN outputs considered; spikes in the output at sharp changes for input, steady state error after the settling of step oscillations, and error in the oscillations during the transient period. All three of these errors occur in Figure 4.2; however, it can be difficult to see these errors for certain plots, or in contrast these errors can be so large it becomes difficult to analyze the output. Therefore for the majority of comparisons analyzed, the absolute values of the output error are plotted. For the remaining Chapters, the term error implies the

absolute value. Figure 4.3 shows the error plot for the outputs of Figure 4.2; the overall error of Figure 4.3 is approximately 5, which is the summation of the error for all time steps in Figure 4.3.

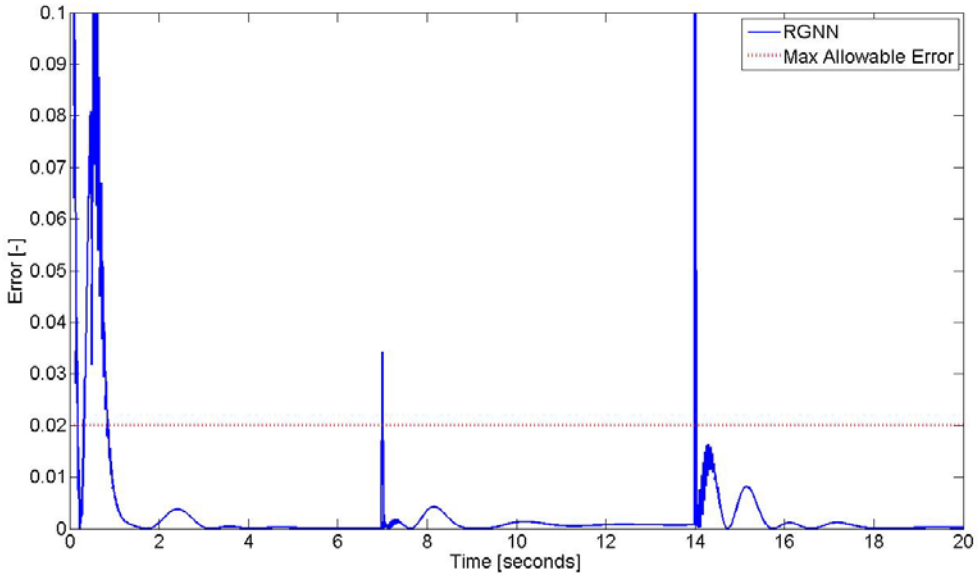


Figure 4.3: Error comparison of output shown in Figure 4.2 for [-5, 5] initial weight interval for "all possible" weights trained.

The robustness of training a RGNN using CAs was tested by limiting the number of synaptic weights being optimized during the training process. Figure 4.1 showed that only eight weights in the matrix were not equal to zero; if a synaptic weight value is zero, mathematically that means there is no connection. Because only eight non-zero weights are present, only the "required" weights are needed to properly represent the test model.

The first step test conducted to evaluate the robustness of an RGNN trained using the CA was to set all unnecessary connections to zero and distribute the weights needed between [-5, 5]. After training multiple networks using these criteria it was found that the resulting network error improved; the error of each network trained for a specific case was averaged; the error was found to decrease as the number of random connections decreased. But even with the decrease in the weights being optimized, no acceptable

networks were achievable. Figure 4.4 shows the error of a network trained using all possible weights and the required weights compared to the desired output.

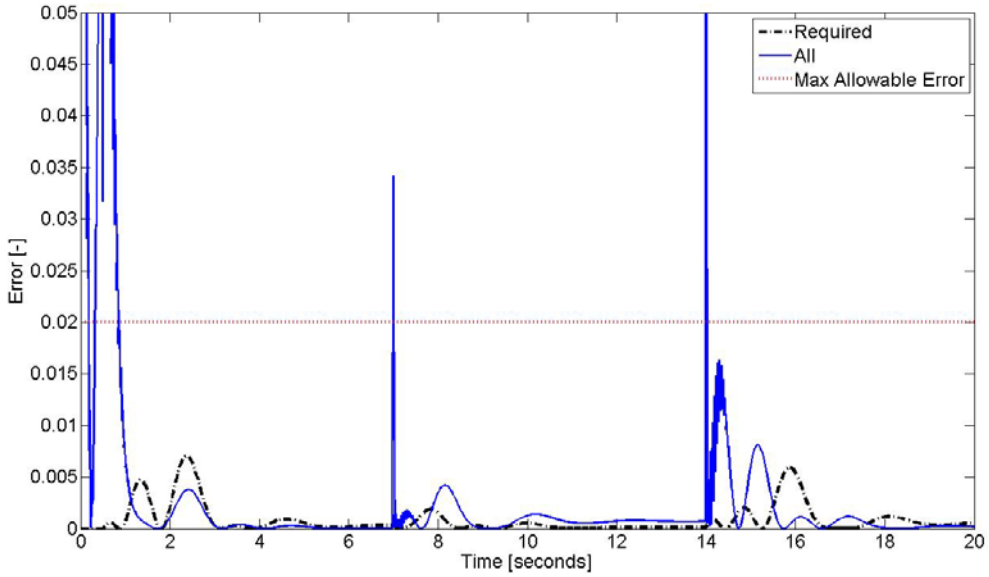


Figure 4.4: Comparison of outputs for desired system, RGNN with all weights trained, and a RGNN with only necessary neurons trained.

The decrease in the number of weights being optimized in the network led to the decrease in error, as seen in Figure 4.4. However, this decrease was found to be insufficient to be classed as a properly trained network as the error did not achieve the acceptable maximum error. Since randomly initializing only connections necessary was found to be ineffective for finding a valid network, the "approximate" network shown in Equation 4.3 was changed one weight at a time to study the effectiveness of the CA. Figure 4.5 shows the error for a network trained with the connection between the input neuron and the third neuron, w_{31}, being optimized.

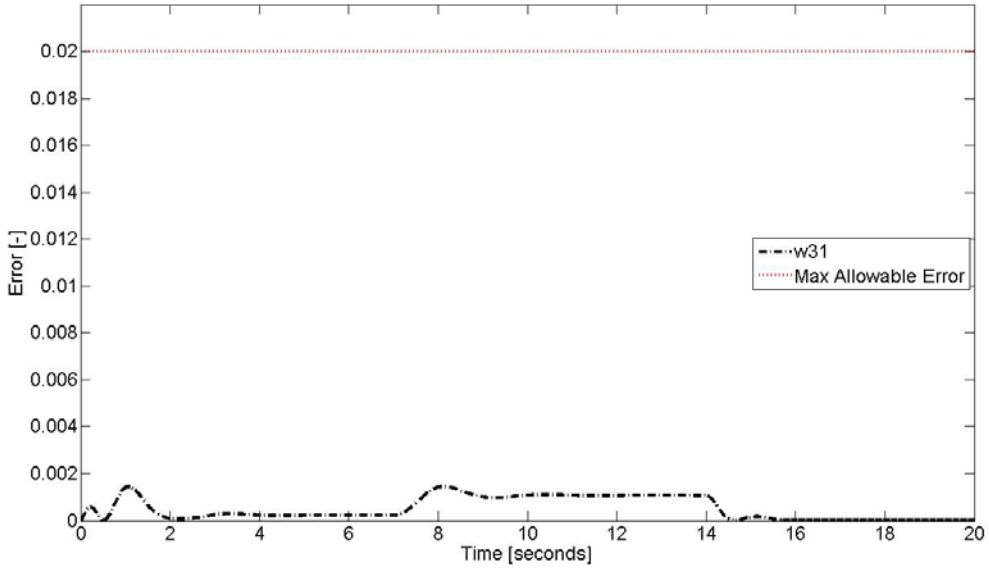


Figure 4.5: Error comparison between the RGNN trained for w_{31} and the desired output using the CA for a step input.

For almost all neurons with a non-zero weight connection, a RGNN was trained using CAs which met the maximum error requirements; it was found that almost all networks produced a similar result to Figure 4.5 by creating an error of less than 0.02 which as mentioned earlier was the maximum allowable error. It was found that for varying the feed-forward and delayed feedback connections between the fourth neuron and the output neuron, w_{46} and w_{64}, the network was not able to reach the maximum allowable error. Figure 4.6 shows the error comparison results when w_{23} and w_{31} were optimized individually and Figure 4.7 shows the error comparison when w_{46} and w_{64} were optimized individually. The error results were separated into two different figures due to the drastically different error results.

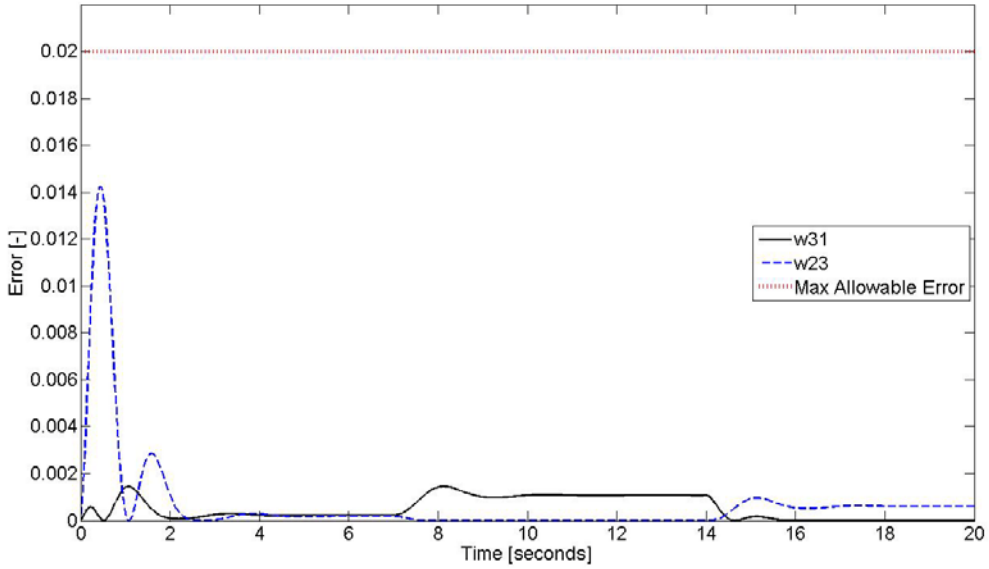


Figure 4.6: Comparison of error results for networks trained by optimizing w_{31} and w_{23} (one weight at a time).

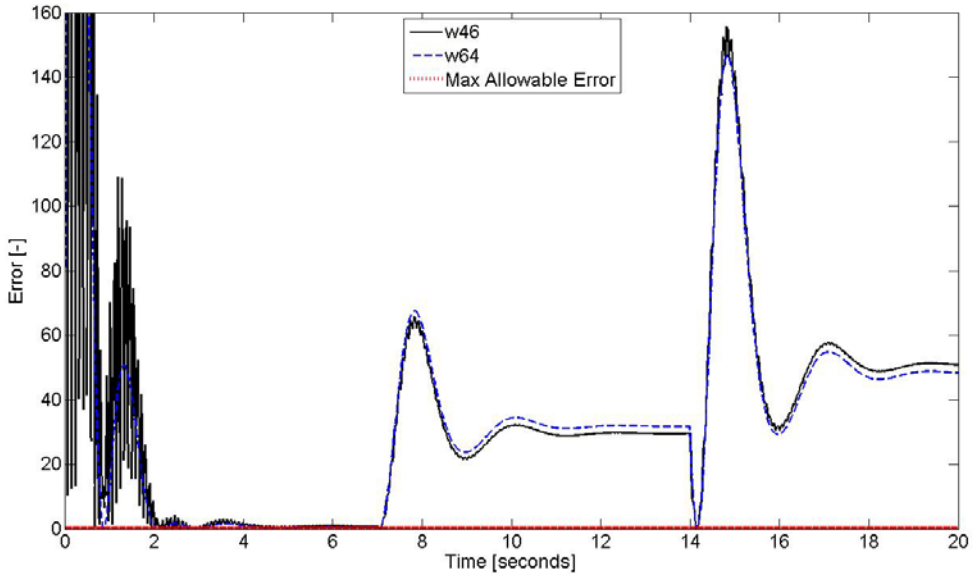


Figure 4.7: Comparison of error results for networks trained by optimizing w_{46} and w_{64} (one weight at a time).

As stated above and shown in Figures 4.6 and 4.7 for optimizing only one weight, for the majority of weights an acceptable solution was obtained when only one weight was being optimized. However, when w_{64} or w_{46} were trained on their own, large training

errors were obtained. The next test was to increase the number of weights being optimized at the same time from one neuron to two neurons. The neurons were optimized for six different weight pairings; w_{31} and w_{64}, w_{31} and w_{46}, w_{46} and w_{23}, w_{56} and w_{42}, w_{46} and w_{64}, w_{45} and w_{31}. These connections were chosen because they cover a broad range of combinations which include both static and dynamic connections.

As was done for optimizing only one weight at a time, an interval of [-5, 5] was used to select the two initial weights of the system to be optimized. It was found that when a weight – which trained properly on its own – was trained in tandem with either w_{46} or w_{64} – both of which did not train properly using step inputs – the error increased drastically. However, for pairs which did not contain w_{46} or w_{64} as a training variable, the results were very similar to those obtained by optimizing only one weight at a time. Table 4.1 also shows that for pairings including w_{46} and w_{64} – even when paired up together – the error was lower than w_{46} or w_{64} on their own. These results show that the resulting error is dependent not only on the number of neurons being trained, but also which specific neurons are being trained. It should be noted that the average time column located in Table 4.1 is used solely for comparison reasons. Also, the average minimum error shown in Table 4.1 is the average error of the networks trained for the specific case listed. The training time is dependent on a variety of things included the processor in the computer used for training and the coding of the algorithm. Computers with identical hardware were used for the training of all networks presented.

Table 4.1: Error results for RGNNs trained using one and two weight optimization with a random initial population.

Weights	Average Count	Average Time (minutes)	Average Minimum Error
w_{23}	1804	1.695	0.00647
w_{64}	34011	26.580	243.67964
w_{46}	3131	16.313	391.52019
w_{31}	1846	1.676	0.00539
w_{56}	1751	2.198	0.00493
w_{42}	2264	2.641	0.00594
w_{31} w_{46}	3616	17.436	92.94014
w_{31} w_{64}	11573	23.483	45.89813
w_{46} w_{23}	3719	16.021	95.88282
w_{56} w_{42}	3216	7.193	48.47830
w_{46} w_{64}	36017	36.669	209.79797
w_{45} w_{31}	2473	2.850	0.00576

Table 4.2 contains a summary of average errors obtained for the different types of testing conducted for the initial weight distribution between [-5, 5]. Additionally, Figure 4.8 shows an output comparison for all types of trained test cases on the [-5, 5] interval. Figure 4.8 only shows the first step in the multi-step input; this is done to show the dynamic effects of the step change in greater detail.

Table 4.2: Summary of average error results for networks created with an initial [-5, 5] distribution.

Weights	Average Count	Average Time (minutes)	Average Minimum Error
Single	7468	8.517	105.87043
Double	10102	17.275	82.16719
Required	36150	38.183	8.11866
All	43623	34.573	17.49052

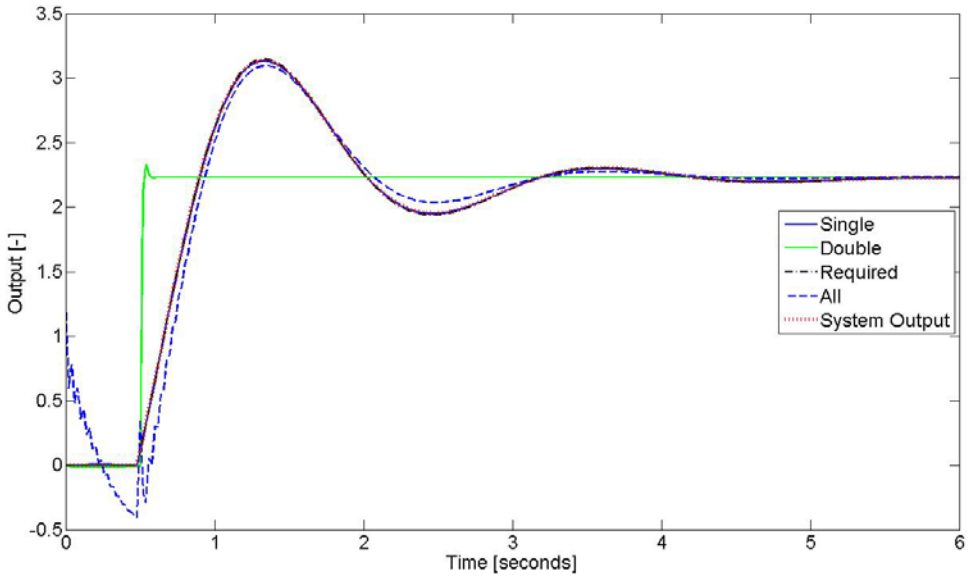


Figure 4.8: Comparison of outputs for all conducted test cases using a [-5, 5] initial weight interval (note that the single weight and required weight cases lie on top of the desired output).

As shown in Table 4.2 and Figure 4.8, the reduction in the number of weights being optimized does not necessarily guarantee that the resulting error will decrease. These results are counter intuitive as it is expected that for any optimization process – neural networks or other types of systems – as the number of variables decreases, the achievable accuracy decreases. The results do show that the ability for a RGNN to optimize a specific weight is dependent on both the number of weights being optimized, and which specific connections are being solved for. Due to the findings that a decrease in the number of weights being optimized during training does not necessitate a decrease in error, the next testing conducted was to decrease the interval size from which the initial population was created.

4.3.2 Training a RGNN Using Step Response with a Limited Initial Population

To test the robustness of training RGNNs using CAs the next step was to decrease the initial population interval to increase the chances of error reaching the prescribed

maximum allowable value. The chances of reaching the global error minimum is assumed to increase as the initial population interval decreases because the number of local minimums should decrease and probability of more initial population points landing on the global minimum upon the initialization of the population increases due to the smaller sample space. For the random initialization of weights discussed in the above section a reference to the exact solution was not necessary; the test range of [-5, 5] encompassed the maximum and minimum values for the exact weight solution. When the perturbation from the initial population becomes limited, attention must be paid to ensure the exact values still fall within the initialization range. Two perturbation ranges were considered for testing the ability of CAs to train RGNNs as the population interval decreases; [-1, 1] and [-0.5, 0.5]. These ranges do not refer to the range in which the weights lie, but the range from the exact solution shown in Equation 4.3 which the weights may deviate from.

Equation 4.3 shows that for an initial population of simply [-1, 1]; w_{23}, w_{46}, w_{56} and w_{64} are all either outside of this range or at the very limits. To ensure a correct initial population was created, the weights were initialized in the following manner,

$$W_k^{init} = W_k^{exact} + \Delta W_k . \qquad \textbf{(4.4)}$$

In the above equation k refers to the individual being changed, W^{exact} is the exact value for the weight matrix given in Equation 4.3 and ΔW_k is a random value matrix where each cell is between the perturbation limits specified by which the weights were changed.

Tables 4.3 and 4.4 show the results for all test intervals with all connections initialized and only the required connections initialized respectively. The "File Number" (or "trials") in both tables refers to the training attempt. As mentioned earlier, each RGNN type was trained multiple times; for the cases listed below training was completed using three different initial populations. Figures 4.9, 4.10 and 4.11 show the output results for all networks trained at intervals of [-5, 5], [-1, 1] and [-0.5, 0.5] respectively; output response as opposed to error response was used because error difference between trials was very large and become difficult to observe differences on the same plot.

Table 4.3: Comparison of average error results for [-5, 5], [-1, 1] and [-0.5, 0.5] for all connections initialized.

Interval	File Number	Count	Time (minutes)	Minimum Error	Average Count	Average Time (minutes)	Average Minimum Error
[-5, 5]	1	30870	31.845	34.87055	43623	34.573	17.49052
	2	50000	36.344	12.72642			
	3	50000	35.530	4.87458			
[-1, 1]	1	49145	51.252	21.75547	43349	43.175	7.76833
	2	32237	32.632	0.51390			
	3	48666	45.642	1.03563			
[-0.5, 0.5]	1	50000	36.044	0.08004	41516	40.891	1.22857
	2	49629	53.324	0.02616			
	3	24918	33.305	3.57952			

Table 4.4: Comparison of average error results for [-5, 5], [-1, 1] and [-0.5, 0.5] for only required connections initialized.

Interval	File Number	Count	Time (minutes)	Minimum Error	Average Count	Average Time (minutes)	Average Minimum Error
[-5, 5]	1	25375	27.533	0.00832	36150	38.183	8.11866
	2	33076	34.807	0.00500			
	3	50000	52.209	24.34266			
[-1, 1]	1	30869	51.420	23.47027	20178	27.335	7.82535
	2	6781	7.381	0.00295			
	3	22884	23.204	0.00282			
[-0.5, 0.5]	1	4298	3.477	0.00410	10035	7.617	0.00473
	2	5937	4.602	0.00601			
	3	19870	14.771	0.00406			

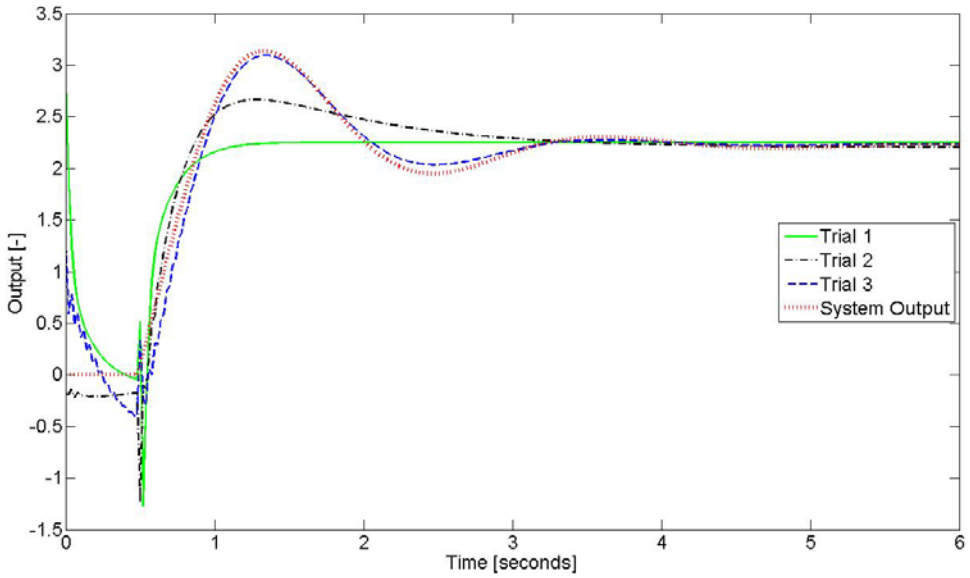


Figure 4.9: Output results of RGNNs trained for [-5, 5] with all weight connections optimized.

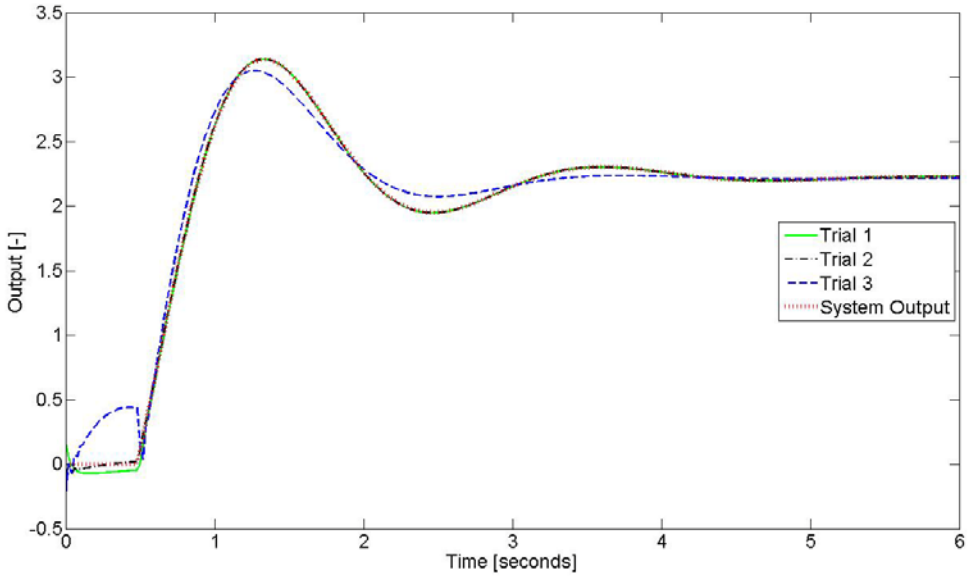


Figure 4.10: Output results of RGNNs trained for [-1, 1] with all weight connections optimized.

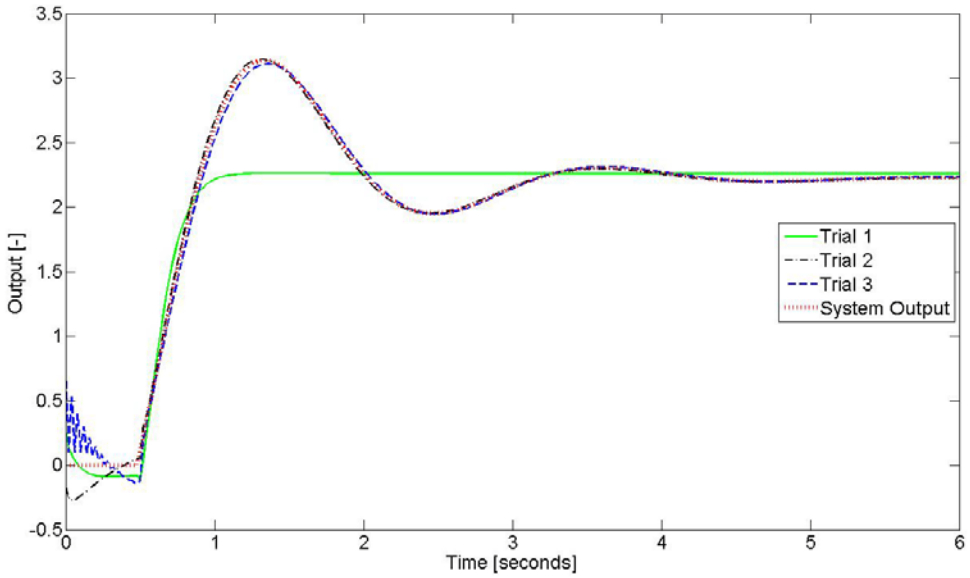


Figure 4.11: Output results of RGNNs trained for [-0.5, 0.5] with all weight connections optimized.

As seen in Tables 4.3 and 4.4, when the perturbation interval decreases the average error also decreases. Figures 4.9, 4.10 and 4.11 show that although one trained network may bring up the average error, as the interval decreases the superior networks for each interval also improve. Although the steady state values are all identical, the most discernable features between the three figures are the improvements of the dynamic features of the step response; these features are seen as the step occurs and ends when the steady state is reached. As the interval was decreased, the dynamic features become similar to the desired output as the oscillations begin to overlap.

4.4 Frequency Based Response Training Using CA

One input type not often considered in training and testing neural networks is the use of frequency based response approaches. But as stated by Lamontagne [2001], one of his conclusions explaining the lack of accuracy in his neural network was an incomplete training set. Although Lamontagne did not attribute his incomplete data set to a lack of frequency based data – in fact he stressed the importance of both magnitude and frequency data – for most neural network research applications frequency based response

data approaches are often overlooked. Most often only step response or random inputs are considered for the training of neural network systems. The following section will focus on using frequency based response data to train a RGNN using the CA for the system previously considered for step response. For clarity purposes, the frequency based response is the output response of the discrete model to a series of input sinusoids (shown in Figure 4.12). The frequency based training data (which contains both input and output data) contains three different frequencies; the frequencies are 0.2 Hz, 1 Hz and 5 Hz. It must be noted that the frequency based response can be considered a limited version of the classical frequency response which contains a large bandwidth of frequencies. However, a full range frequency response was not used as initial feasibility studies with swept sinusoids resulted in unstable training results and hence a swept signal was not further pursued.

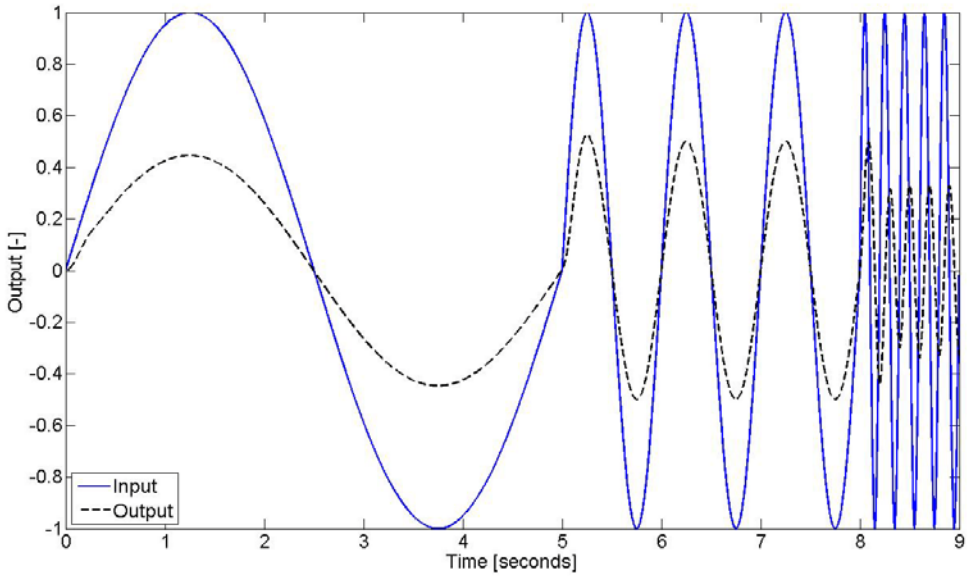


Figure 4.12: Frequency based training data input and output data.

As was done in the previous section a variety of connection morphologies were tested for frequency based inputs. Unlike the use of step response training previously discussed, RGNNs for the most part were only trained using the [-1, 1] initial population interval rather than random initial values or the smaller [-0.5, 0.5] population interval. The narrow focus towards one interval was done because the basis of frequency based

response is to study the difference between the step and frequency based responses, inconsequential of the interval. The [-1, 1] interval was chosen because for step responses the interval showed an improvement in the dynamic capabilities of all test cases conducted. However, in turn it also afforded room to improve over the step response results; the same cannot be said for the [-0.5, 0.5] interval.

As mentioned above, the [-1, 1] interval was used for the most part for the training of network using frequency based response. The [-5, 5] interval was used to test the differences between step and frequency based response for the training of w_{46} and w_{64} individually. This is due to the inabilities for step response training to work at the [-5, 5] interval for these weights; that is, it was of interest to see if the frequency based inputs could produce acceptable results. The [-1, 1] interval was used to train networks for the following weight connection types; two connections, only required connections and all connections.

The first test conducted was to observe possible improvements to the optimization when training either w_{46} or w_{64}. Table 4.5 shows the comparison of training either w_{46} or w_{64} individually using step and frequency based response training. Table 4.6 shows the comparison of training two weights at a time using frequency based training and step response training. It should be noted that the actual value of the average error cannot be compared because different inputs were used. In general, it was observed that the trends of results for frequency based responses show a vast improvement over the trends of the step response training.

Table 4.5: Trend comparison of step and frequency based training for w_{46} and w_{64}.

Weight	Training Method	Average Count	Average Time (minutes)	Average Minimum Error
w_{46}	Step	3131	16.313	391.52019
w_{46}	Freq. Based	2166	5.252	0.00565
w_{64}	Step	34011	26.580	243.67964
w_{64}	Freq. Based	3028	7.266	0.00387

Table 4.6: Trends comparison results of step and frequency based training when two connections were trained in tandem.

Training Method	Weights	Average Count	Average Time (minutes)	Average Minimum Error
Freq.	w_{31} w_{46}	4270	40.687	24.50049
	w_{31} w_{64}	32877	90.537	22.95698
	w_{46} w_{23}	5154	36.174	24.50049
	w_{56} w_{42}	4570	41.679	33.20162
	w_{46} w_{64}	8618	30.312	8.17109
	w_{45} w_{31}	4112	46.877	37.55219
Step	w_{31} w_{46}	3428	17.147	88.48946
	w_{31} w_{64}	42349	35.400	61.25802
	w_{46} w_{23}	3567	15.330	89.84444
	w_{56} w_{42}	2958	5.770	48.47782
	w_{46} w_{64}	21985	18.984	132.98764
	w_{45} w_{31}	1751	1.573	0.00557

For networks which were trained using different data types, a smaller training error does not necessitate a more generalized neural network. Similar to Tables 4.5 and 4.6 the use of frequency based training produced a smaller training error for almost all cases. For the purpose of the following analysis the "generalization" of network refers to the ability of a trained network to produce a small error (when compared to the input the network was trained for) for an input signal it was not trained for. In the sections above, the networks were trained for a multi-step input and a multi-frequency based sinusoid input separately. It was now of interest to examine how well these trained networks generalized for an input they were not trained for. If the same input was used for both trained networks, then the average error can be used for comparison.

Figure 4.13 shows the error comparison between step and frequency based response for the best trained networks when all connections were trained for a step. As seen in Figure 4.13, the frequency based response trained network produced spikes which go above the acceptable minimum error line located at 0.02. Even with the spikes overshooting the maximum acceptable error, the majority of the frequency based trained network remains under this value and creates a total error or 1.2241. When the same plot

was produced but for a frequency based input the results were the same; the network for which the test input was trained produced good results, while the other network produced relatively good results but with those results not reaching the maximum acceptable error.

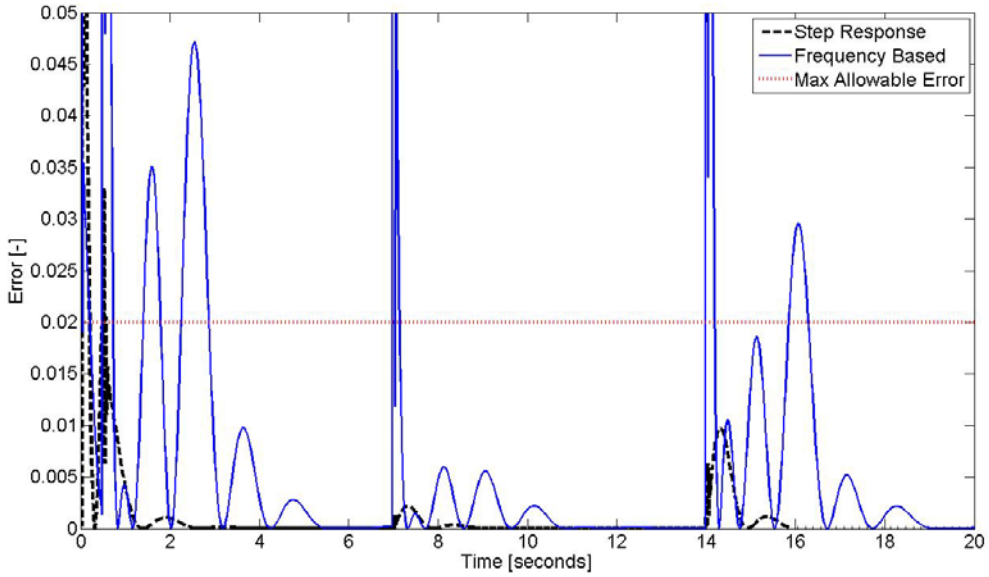


Figure 4.13: Error comparison for step and frequency based response trained networks using all connections for step input data.

Table 4.7 shows the error results for a network trained using frequency based data using a step input and a network trained using step response data using a frequency based input. The last column in the table (Ratio Test/Trained) refers to the ratio of the error produced by a network using the opposite input signal from the input used to the train the network, over the training error for a specific input signal. For example, if a network used a step input signal and had a training error of 40, and if the network was then given a frequency based input which produced an error of 80, then the Test/Trained Ratio would be 2. Conversely, if a frequency based input produced a training error of 200, and a step input produced an error of 100, then the Test/Trained Ratio would be 0.5. If the Test/Trained ratios are compared for the two above examples – based upon the definition for generalization given earlier – the frequency based trained network would be

considered to be more generalized even though the step response trained networks produced lower errors.

Table 4.7: Error comparison between the outputs of RGNNs trained using step response and frequency based training. (Note that the "Opposite Data Output Error" refers to the case where the network was trained with a frequency based input and tested with a step input etc.)

Weights Trained	Training Data	Average Minimum Error	Opposite Data Output Error	Ratio Test/Trained
Require	Frequency	0.00461	0.01474	3.1974
	Step	8.11866	2.96597	0.3653
All	Frequency	37.69136	569.95981	15.1218
	Step	17.49052	190.63298	10.8992

The method shown in Table 4.7 was used because obtaining a properly trained network proved difficult. Using this method allows the comparison of networks, even for cases where poorly trained networks were obtained. The trend of step response based trained networks producing more generalized networks holds true in general for all the networks tested except when w_{46} and w_{64} were trained one at a time as shown in Table 4.5. Both step response and frequency based training produced generalized networks. However, the generalization for networks was not consistent enough to conclude that generalized networks were achievable for all conditions.

4.5 CA Training Using a Combination of Step and Frequency Training Data

Frequency based response training was shown to increase the robustness of training RGNNs using CAs for networks where w_{46} and w_{64} are trained. One drawback to frequency based response training is the lack of direct steady state magnitude variance included in the training data. For linear systems this aspect becomes negligible, but for nonlinear systems the lack of steady state magnitude training data can prove costly. In addition, the sharp edges of step inputs do contain frequency information perhaps not as

distinct as direct frequency (sinusoidal) data. As the global objective of this research was to obtain an accurate neural network model for load sensing pumps, the lack of direct steady state magnitude data could be problematic because load sensing pumps are nonlinear. This is also one of the reasons why multi-step response is common for the training of neural networks.

As a first step, it was of interest to see if a network trained with one type of input could be further trained using a second type of input. The final population of weights were taken after frequency based training was finished and were used as the initial population for training an RGNN using the CA with a step response training data set (as discussed below, this is defined as "two stage" training). Figure 4.14 shows the improvements made using the two stage training strategy. Although there are spikes at the beginning of the training set and at step changes, the spikes are significantly narrower and the total error is now 0.0251; which is 4.3% of the original error, 1.2241. Figure 4.15 shows the error response for a network trained using two stage training when a frequency based input is used. The error of the step response trained network was 0.0396 while the error of the two stage trained network was 0.0327; the two stage trained network shows a 17.2% decrease in error when compared to the step response trained network.

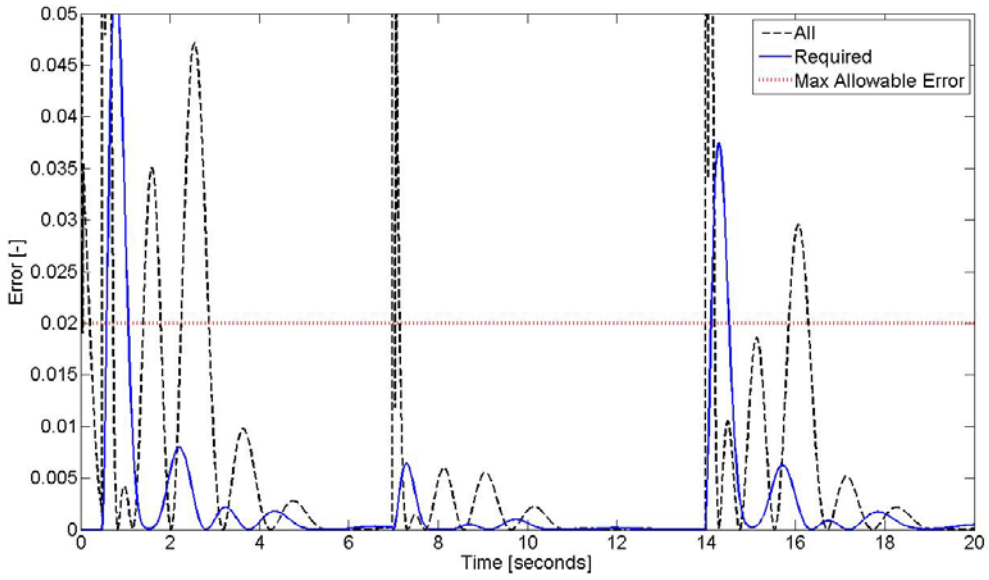


Figure 4.14: Error comparison for multi-step input between frequency based and step-frequency based trained RGNNs.

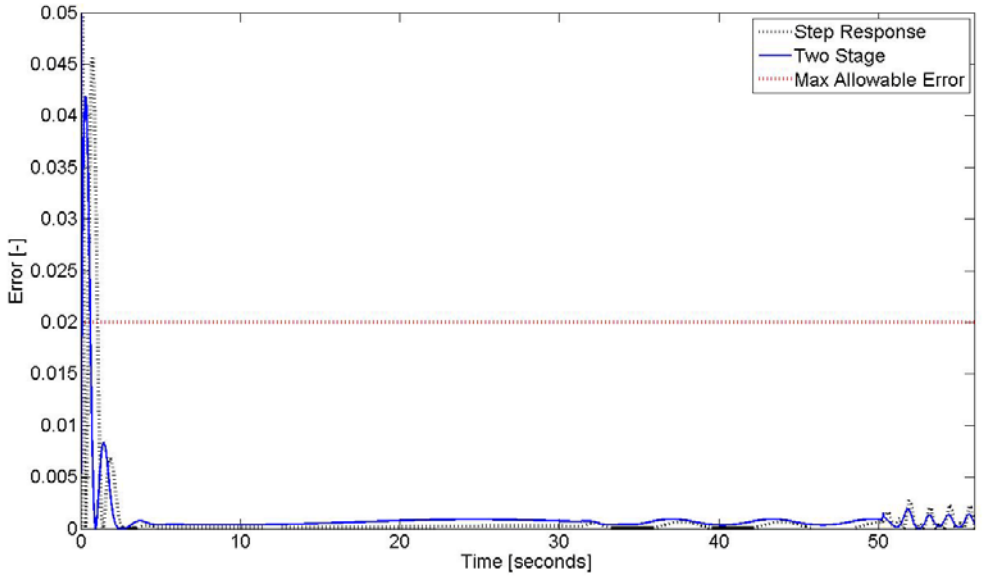


Figure 4.15: Error comparison for frequency based input between step response and step-frequency based trained RGNNs.

Based on the improvements made by combining both step and frequency based training data, two different modified training sets were proposed. The first was a training

set which consisted of the frequency based data directly followed by the step response data all in the same training set; this will be referred to as “single stage” or “one stage” training. The second proposed training set consisted of training the network as two separate training files (the approach used to generate the results shown in Figure 4.14). First the network was trained using frequency based response. When either the allowable time ran out or the maximum allowable error was achieved the training was stopped and the final population of weights was saved. This population was used as the initial population for the training of an RGNN using step response based training; this will be referred to as “two stage” training. The same initial weights were used for both training methods.

Figure 4.15 shows the error comparison for the case where all possible weights were trained using frequency based, one stage and two stage training. It shows the step response error for the most robust one and two stage sine-step trained networks in addition to the best frequency based trained network. It can be seen that the addition of step response training to the frequency based training data set increased the error. Although there was no improvement by expanding the type of training data used to include two different data types, the two stage training proved to have a smaller error when compared to the one stage training method but inferior to frequency based data or step response data alone.

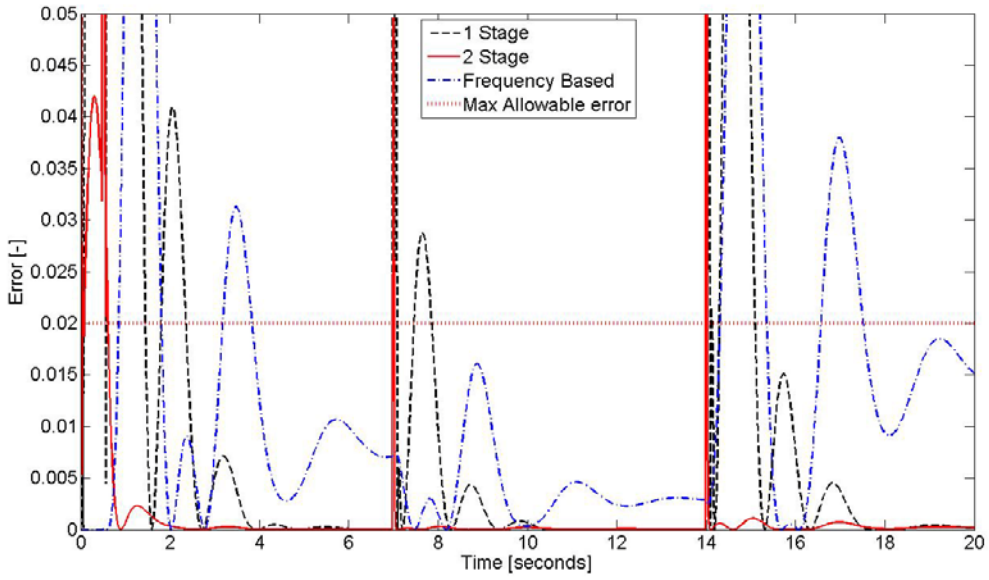


Figure 4.16: Step response for one stage, two stage and frequency based training.

4.6 Summary of CA Training

The concept of applying non-gradient training algorithms has been discussed and shown to have distinct theoretical advantages; however, the application of CAs to RGNNs has unexpected drawbacks. The most common training data used for training neural networks is step response; however, it was shown that step response training could not achieve an acceptable training error for many of the weight combination and interval tests conducted. The most notable finding from step response training using CAs was that the robustness of the network was dependent not only on the interval by which the initial population was made and the number of weights being optimized, but was also dependent on which weights were being trained. There was significant error for all RGNNs trained using the CA with a step input.

To improve on the results obtained using step response training, a frequency based input was used for training RGNNs using the CA. It was shown that frequency based training showed produced a smaller training error when compared to step response for almost all weight combinations used. It should be noted that these errors were found

using different input signals, so they are not directly comparable. These results were most notable when weights w_{46} and w_{64} were involved, as they proved troublesome for step response training. When the networks were tested to see which training method produced a more generalized network it was found that step response training for most cases produced a more generalized network for most cases, but there was not a strong enough trend to conclude that step response training produced a more generalized network overall.

As suggested by Lamontagne [2001], a key aspect to the training of neural networks is the training data used. As it was shown, the inclusion of frequency based training improved the error response when compared to RGNNs trained using step response. Therefore to create a more complete data set, step and frequency based inputs were used in two different methods. First the two were combined into one training set and RGNNs were trained with all weights being optimized. Second, the training process was broken up into two processes; training using frequency based response and then using the final population to begin training using step response training. It was shown that although the two stage training process had a lower output error compared to the single stage training process, both methods proved to be less generalized when compared to step response only training or frequency based only training.

Now that the use of CAs to the training of RGNNs has been explored for a variety of conditions, the next section will consider the same conditions but using a different training algorithm. Chapter 3 described the use of the GA which – like the CA – is a non-gradient based algorithm; the GA will be used next to study the robustness of CAs when compared to the GA which is a more traditional non-gradient training method.

Chapter 5: Comparison of Complex Algorithm and Genetic Algorithm

5.1 Introduction

In Chapter 4 a sensitivity study was completed which considered the use of CAs for training RGNNs. The training of multiple connection methodologies showed that for the many cases, CAs were effective for training RGNNs to model a second order transfer function. The second objective of this thesis was to conduct a comparative study of CAs to a commonly used non-gradient training method known as the genetic algorithm (GA). The following chapter will be used to show the advantages and disadvantages of this training method compared to the CA. To do so a comparison will be made to CAs using the same morphologies and initial population intervals studied in Chapter 4, but instead trained using the GA.

Chapter 3 discussed the GA and gave an algorithm, along with corresponding equations needed to train a neural network. For the following discussions two changes have been made to the algorithm for the purposes of this thesis. The first change involves an altering to the mutation algorithm which in Chapter 3 was given by,

$$w^{i(mutate)} = w^i + \left(b - w^i\right) f\left(G\right), \qquad \text{If } r_1 < 0.5 \tag{5.1}$$

and

$$w^{i(mutate)} = w^i - \left(a + w^i\right) f\left(G\right), \qquad \text{If } r_1 \geq 0.5. \tag{5.2}$$

In the equations above the mutated weight is limited to the high and low limits of the training interval. This is intended for systems where the weight is known to lie in a specific interval or is forced to lie in a certain interval.

For the system being modeled the weights are known as previously discussed in Chapter 3; however, for a neural network to train properly, the training algorithm must be allowed to change the weights as desired without restriction. Although in Chapter 4 the initial population was created around a certain interval, the weights were free to move

outside of this interval in order to reach a maximum allowable error. Therefore, to allow the weights to mutate freely the mutation process was altered to,

$$w^{i(mutate)} = w^i + r_1 w^i \left(-1^{r_2}\right) f(G), \quad \textbf{(5.3)}$$

where r_1 is a random value between [0, 1] and r_2 is either 0 or 1. By doing this the value of the new weight is not limited to a specific interval.

The second change is based upon some initial findings from the training of RGNNs using the GA. The initial findings showed that using the GA, RGNNs had difficulties producing stable inputs due to unstable dynamic weights. The GA is designed to change the weights of the network based upon the genetic information of the parents as discussed in Chapter 3. A restriction placed on the algorithm was to conduct the crossover sub-loop ten times if the previous iteration produced children who were worse than their parents. It was observed that even when such a restriction was placed no suitable children could be found. Furthermore, based on the algorithm the children were simply placed into the next population once the maximum sub-loop iteration was reached even when the children produced a higher error than their parents. This meant that the new population had the potential to be worse than the old population.

To alleviate this occurrence a restriction was placed on the algorithm which stated that the new population cannot be worse then the old population. This means that if one of the children has a higher error than either parent, then the child is not placed back into the population; instead the parent is. Although this means the population can become oversaturated with the same genetic information, this also means the RGNN can never get worse.

5.2 Step Response Training of a RGNN Using GA

As mentioned in Chapter 4, multi-step response training is very common for the training of neural networks due to the inclusion of both frequency and magnitude data. Using GAs, RGNNs were trained with step response data for the same morphologies as

CAs. Chapter 4 showed that the initial population contributed to the training ability using the CA; for some cases an acceptable error was obtained, but even when the same weights were changed and same initial population interval was used, another training attempt did not achieve the same error. Therefore, for the application of GAs the same initial population data (the initial weights) were used to train RGNNs using GAs so an accurate comparison of training capabilities could be completed.

The first connection morphology considered was the exact weight matrix required (see Equation 4.1) with only one synaptic weight being trained for. Table 5.1 shows the results for the number of training cycles (iterations) used, the average minimum error and the training time needed for weights w_{23}, w_{64}, w_{46}, w_{31} when trained individually; both the CA and GA results are presented. The weights shown were chosen because they represent connections in the input and output paths in addition to delay and no delayed paths. Note that because the same multi-step input was used for both cases, the average error can be compared

Table 5.1: Comparison of results for training the training of one weight with [-5, 5] interval.

Weight Trained	Training Algorithm	Average Count	Average Time (minutes)	Average Minimum Error
23	GA	168.3	60.211	8443.35719
	CA	1804.0	1.695	0.00647
64	GA	166.7	60.152	367.00920
	CA	34011.0	26.580	243.67964
46	GA	166.3	60.134	393.97909
	CA	3131.0	16.313	391.52019
31	GA	167.7	60.188	3979.75479
	CA	1846.3	1.676	0.00539
56	GA	167.7	60.221	680.99781
	CA	1751.0	2.198	0.00493
42	GA	397.7	60.134	4721122.94769
	CA	2264.3	2.641	0.00594

Comparing the results for GAs and CAs in Table 5.1, it is seen that the GA has a considerably larger error for all single weight trained networks; this trend will be

discussed later. Figure 5.1 shows a typical error signal for a single weight trained network using the GA. As mentioned in Chapter 4, the maximum allowable error for a RGNN to be considered properly trained is 0.02; this line is shown at the bottom of Figure 5.1. It can clearly be seen that the error for a RGNN trained using GAs for the optimization of w_{31} is typically larger than the allowable maximum error for the training data set; the total output error for Figure 5.1 is 822; this was obtained using the error calculation methods shown in Chapter 3.

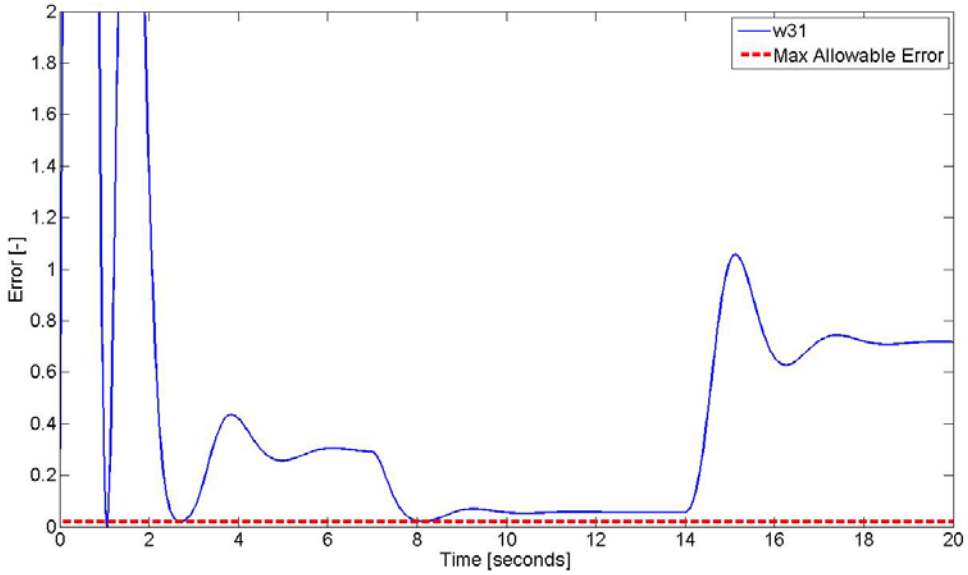


Figure 5.1: Typical error signal for step input trained RGNN using GAs when only w_{31} was optimized.

As done for testing CAs in Chapter 4, the next step was to train RGNNs by changing two weights in tandem. Figure 5.2 shows the error signal for w_{42} and w_{23} trained in tandem using the GA; it should be noted that this pairing produced the smallest output error of all trained pairings with an error of 108. Unlike Figure 5.1 which shows curved lines with small oscillation frequency, the error of the trained network shown in Figure 5.2 has large high frequency spikes. The trained network with output error shown in Figure 5.1 has a similar natural frequency to the system being modeled. This is not the case for the natural frequency of the trained model whose response is shown in Figure

5.2. However, the error of the single weight trained network is almost eight times higher than the two weight trained network.

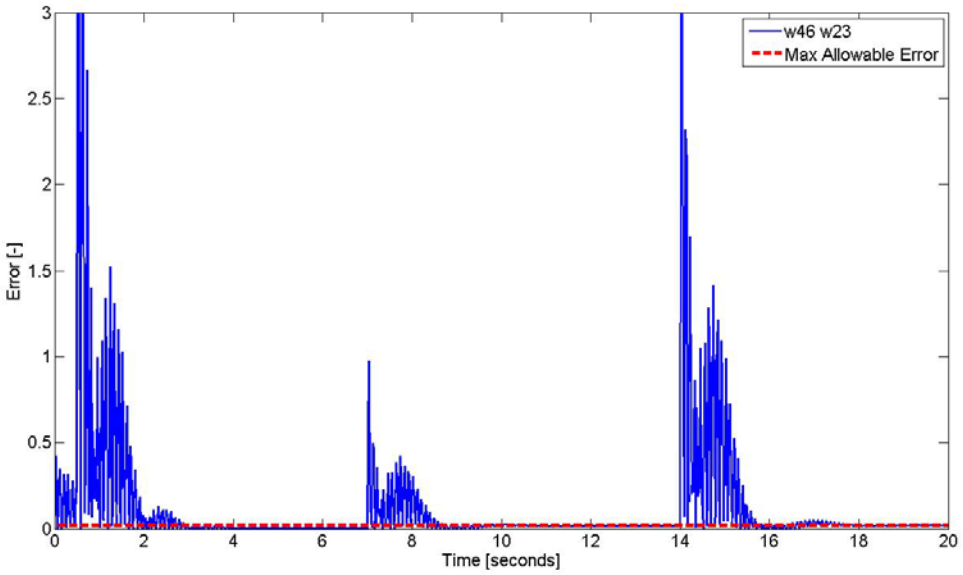


Figure 5.2: Error signal for step input trained RGNN using GAs when w_{46} and w_{23} were trained in tandem.

This is the opposite of what occurred when training RGNNs using CAs; as the number of weights being trained increased from one to two, the error increased significantly. For GA trained networks as the number of weights increased the error decreased. To test whether there is a link between the increase in neurons being trained and a decrease in output error when using GAs, networks were trained using only the number of required neurons and the training of all possible neurons as conducted in Chapter 4 when examining CAs. Figure 5.3 shows the error comparison between training cases containing two weights, only required weights and all weights for the first step of the training signal.

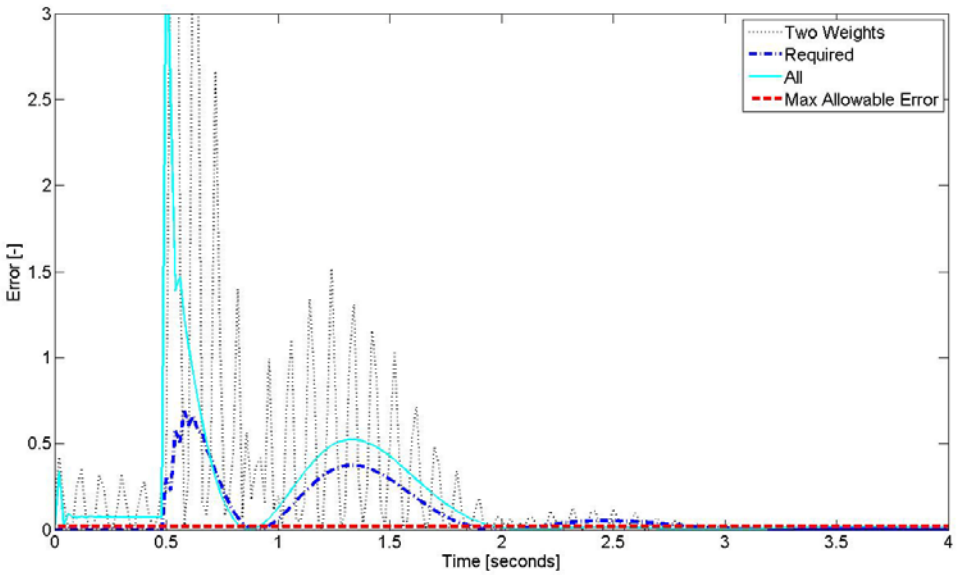


Figure 5.3: Error comparison for networks trained using one weight, two weights, required weights and all weights.

Figure 5.3 shows a smaller error signal when only the required weights and all possible weights are used when training a network compared to when only two weights are trained. Figure 5.4 shows the output response for networks trained for two weights, only the required weights, and all weights; it should be noted that Figure 5.4 only shows the first step in the multi-step input. The natural frequency of the RGNNs did not improve when the number of neurons increased. When the number of neurons increased the oscillations disappeared at the step points creating no oscillations at sharp changes to the input.

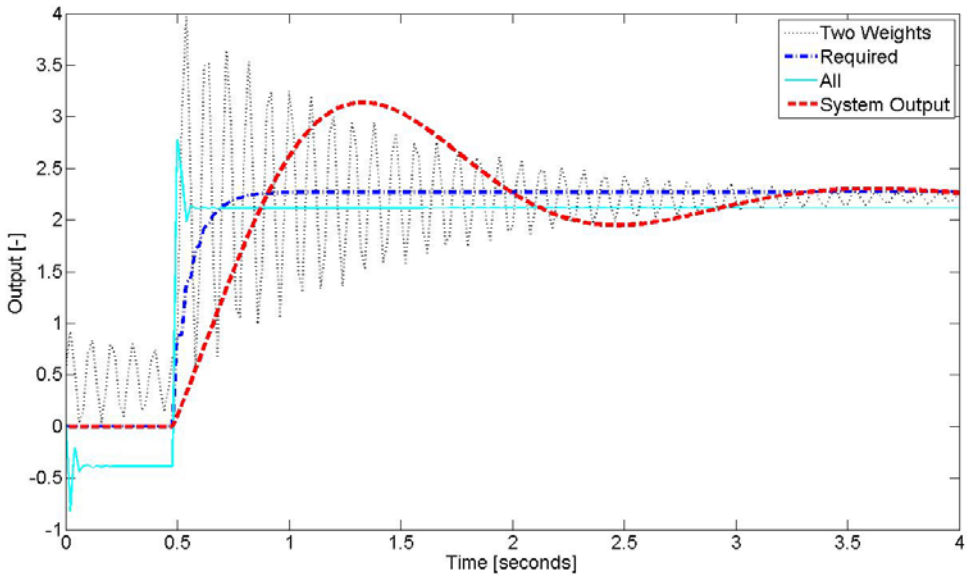


Figure 5.4: Output comparison for networks trained using two weights, required weights and all weights.

Figures 5.1, 5.2 and 5.3 show a distinctive trend when using the GA to train RGNNs; as the number of weights being trained increases the error of the network decreases. This is the opposite what occurred when CAs were used to train RGNNs, and it is also counterintuitive to what is expected to occur. As with CAs, it is expected that as the number of variables – in the case of neural networks, the variables are the weights – decreases, the accuracy of the algorithm should increase.

One possible rationale for this not occurring with GAs is a restriction placed on the training algorithm mentioned at the beginning of this chapter. Because the GA proved ineffective for the training of networks when no restrictions were placed on the creation of a new population, the GA was modified so that the new population would at the very least never have a higher error than the previous population. This stipulation decreased the flexibility of the GA – which is one of the powerful tools of GAs – but was necessary to create stable outputs. The limitation of the new population caused the population to rapidly collapse.

The CA has a modified component to allow the complex to move away from a local minimum; the only comparable mechanism for the GA is mutation. Because the mutation mechanism for GAs is very random, the limitations of the algorithm led to the inability for the GA to find the global minimum. When using GAs, as the number of weights increased, the ability for the population to drift towards a lower error minimum also increased. This is because the increase in the number of weights (variables) increased the amount of genetic information, and as the algorithm has more genetic information to choose from the robustness also increases.

Figure 5.5 shows the error comparison between the CA and the GA when all possible weights of a RGNN are being trained for with an initial population interval of [-5, 5] for the first step in the multi-step training input. For all intervals, the results where similar to Figure 5.5 which compares the GA to the CA using the initial population interval of [-5, 5]. Figure 5.5 shows that the CA has a significantly smaller error compared to the GA for the dynamic portion of the step response, 0-2 second portion of Figure 5.5; the steady state output for both CAs and GAs are similar and both are underneath the acceptable maximum error line.

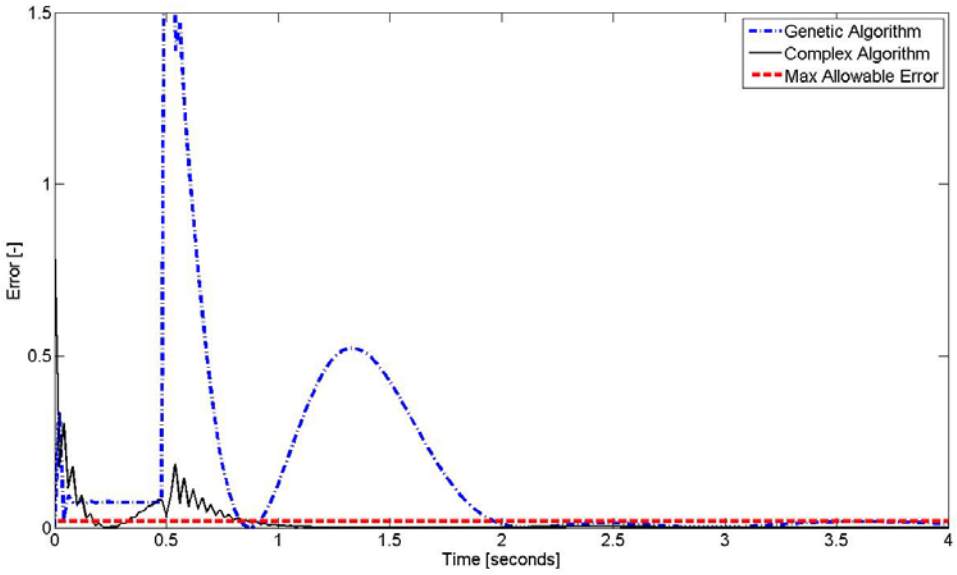


Figure 5.5: Error comparison for CAs and GAs when training all possible weights with an initial population interval of [-5, 5].

It was shown in Chapter 4 that the robustness of training using CAs was dependent on the initial interval size. The same tests were performed using GAs to discover if the dependence on initial population interval was a trend for other training algorithms besides CAs. Figure 5.6 shows the error comparison of all three test intervals for the first step in the multi-step training input. The secondary error peak (at approximately 1.5 seconds) in Figure 5.6 shows that the error of the smallest interval, [-0.5, 0.5], is the largest of the three intervals at that point. This is the only peak where the smallest interval has the largest error contribution. For all other peaks in the error comparison, the [-0.5, 0.5] interval has the smallest error with a value of 24. The largest overall error found for the comparison done in Figure 5.6 is for the [-5, 5] interval with an error of 33, while the [-1, 1] interval has an error of 30. The CA test results found in Chapter 4 show the same trend; therefore it can be stated that as the initial population interval decreases, the error of the trained RGNN will also decrease.

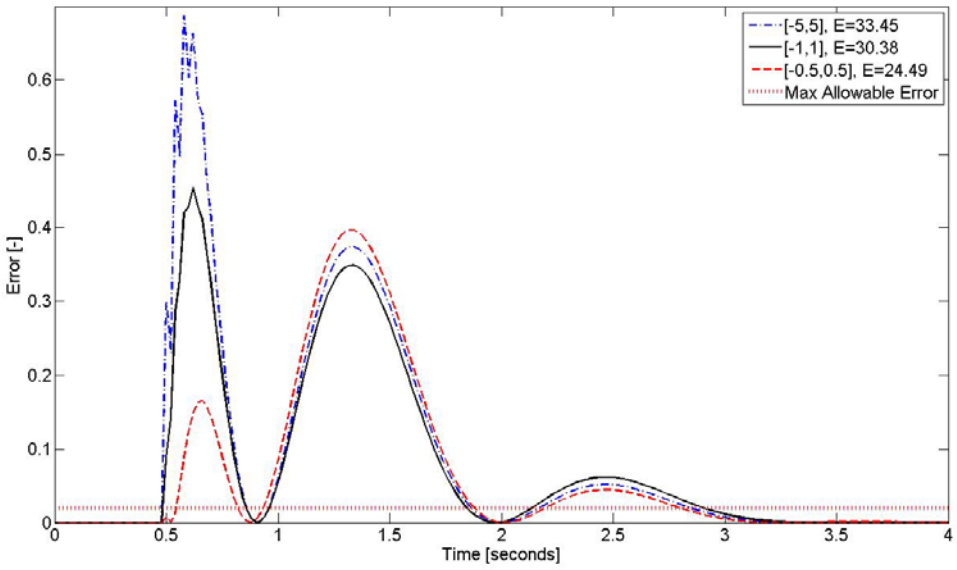


Figure 5.6: Error comparison for initial population intervals [-5, 5], [-1, 1] and [-0.5, 0.5] for RGNNs where only the required weights were trained using the GA.

5.3 Frequency Based Training of a RGNN Using GAs

In Chapter 4 a comparison was done for frequency based training and step response training; it was shown that networks which trained to the acceptable minimum error did not necessarily ensure the trained network was properly generalized for all the cases studied. The use of step response training did in general create a more generalized network than frequency based training, but many of the results were not satisfactory and were not consistent. The following section will discuss the findings for frequency based training of RGNNs using the GA compared to the CA, and will also discuss the difference between the training of networks using frequency based training and step response training. As stated in the previous section, it was found that the GA proved very ineffective for training RGNNs where one weight and two weights were optimized. Because of the inadequacies of these types of networks, only networks where all weights and only the required weights were optimized will be used to discuss the frequency based training of RGNNs using the GA.

Training RGNNs using the GA with step response training data showed a much larger error as was observed in Figure 5.5. Figure 5.7 shows the frequency based error comparison between the GA and CA for frequency based training using the [-1, 1] initial population interval. It shows that during the low and mid frequency inputs the genetic algorithm works well – except for a short initial error spike at the beginning of the input. However, as the frequency increases, the error also begins to increase rapidly; this effect was not observed for the CA. The CA produced error value less than the maximum allowable error for the entire training set as can be seen in Figure 5.7. These results were typical for all networks trained when the training error for CAs and GAs were compared.

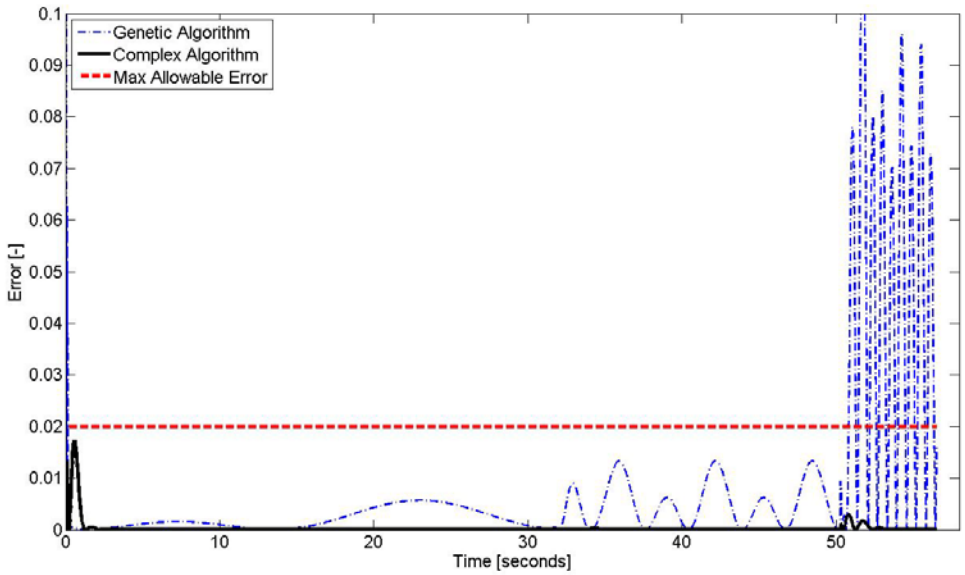


Figure 5.7: Comparison of error between the GA and CA for frequency based training using [-1, 1] initial population interval.

As previously mentioned, Chapter 4 showed the training of RGNNs with the CA using step response data to produce more generalized results when compared to networks using frequency based response training data. Table 5.2 shows a comparison of frequency based and step response training for RGNNs trained using the GA for the same type of inputs. When considering the case where all possible neurons are optimized there is a negligible improvement between the percentage differences when step response training is used; the ratio difference for frequency based training was 3.1 and 3.6 for step response training, this difference is not large enough to make valid conclusions based on GA data alone.

Table 5.2: Comparison of results for the training of RGNNs with GAs using step response and frequency based training for [-1, 1] initial population interval.

Weights Trained	Training Data	Average Minimum Error	Opposite Data Output Error	Ratio Test/Trained
Required	Frequency	19.35809	64.59246	3.3367
	Step	34.5499	17.15014	0.4964
All	Frequency	20.78344	64.87912	3.1217
	Step	48.79635	175.25802	3.5916

Similar to using step response training for CAs, when only the required weights were considered, step response training showed greater generalization. In fact as shown in Figure 5.8, when a frequency based signal was used as the input signal to a step response trained RGNN, the error of the system was lower than the error of a frequency based trained network for the same input. It should be noted that only the last two test frequencies of the training set is shown in Figure 5.8. This is because at lower frequencies the error is very small. The oscillations in error only occur at the end of the data set, therefore the first frequency was omitted to show greater detail for the segments which produced larger errors. The use of GAs has shown that the observations discussed in Chapter 4 regarding different training input signals hold true; step response training has been shown to be more effective then frequency based training.

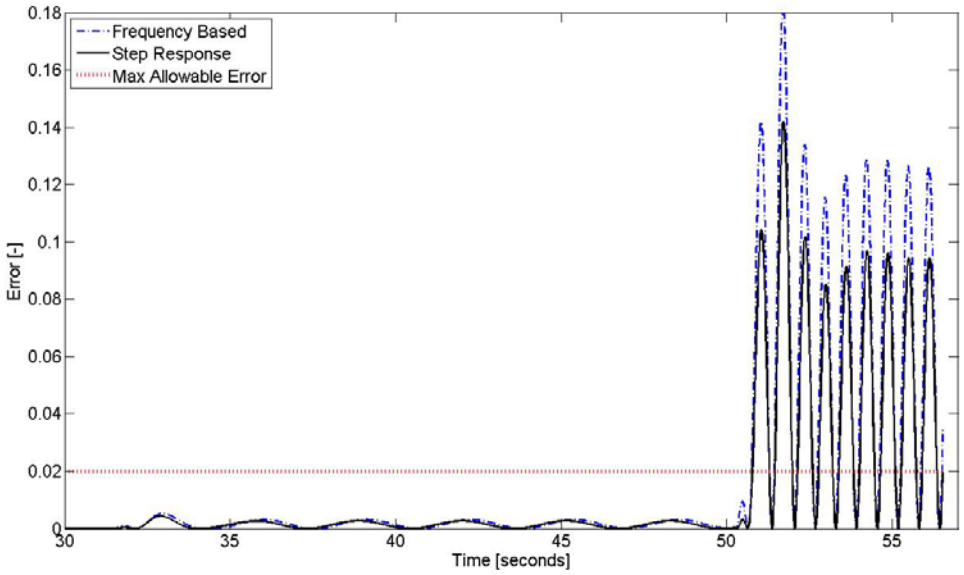


Figure 5.8: Error comparison of frequency based and step response trained RGNNs for a frequency based input signal; only required weights were trained.

5.4 Discussion of GA Results

Testing was conducted using the GA to train RGNNs for the same weight optimization cases and initial population intervals as was done in Chapter 4 using CAs. Because it was discovered when training RGNNs using the CA that the robustness of training a network is dependent upon the initial population, for the networks trained using the GA the initial populations used where identical to those used for the training of RGNNs using the CA in Chapter 4.

Certain modifications were made to the GA after preliminary testing was conducted; these modifications were performed on the mutation algorithm and the crossover portion of the GA. The modifications were done to ensure the mutation weight values could vary outside of the population intervals provided, and also to ensure that the new population of the network was never worse than the old population. It was found that the limitation of ensuring the new population was never worse than the old population created difficulties for the training of RGNNs; this was especially evident for networks where only one weight was trained at a time.

The assurance that the new network never increased in error had an adverse effect; as limits are placed on how the population can mate, the amount of available genetic information decreases. This is why as the number of weights being optimized increased the error decreased; indeed, as the number of weights being optimized increased the genetic information increases. However, adversely, as the number of weights being optimized becomes too high there is too much information and finding the appropriate optimization point becomes difficult. This is the same trend as was found with the CA.

When comparing the results for both step response and frequency response for CAs and GAs it was found that the CA minimized the output error better than the GA. This trend held true for all trained networks tested. One trend which was noticed during the testing of CAs was the increase in robustness of step response training when

compared to frequency based training. This trend held true when step response training and frequency response training was compared for training RGNNs using GAs.

Chapter 6: Conclusions and Recommendations

6.1 Summary of Results

Chapter 4 explored the use of training RGNNs using CAs; the methodology for using CAs was described in Chapter 3. Some of the results showed that as the initial population interval size increased, the ability of the network to obtain an acceptable error diminished. It was also found that the ability for a RGNN to train to an acceptable error using CAs was dependent upon the specific weights being trained.

The training of RGNNs was also used to test the robustness of CAs as the number of weights being optimized was varied. It was found that when only one weight was being optimized – with the other weights were set to their exact values – the majority of networks fell below the maximum acceptable error. However, two connections (w_{64} and w_{46}) proved problematic for the training of RGNNs when using a step input training data set. The hindrance of training capabilities caused by weights w_{64} and w_{46} was also observed during the training of two weights in tandem. As the number of weights being trained increased from two weights to only the required weights needed to model the test system, the robustness of the CA increased. When the number of weights being trained was further increased so all weights were being trained, the error increased; it should be noted the error still remained lower on average than when two weights were trained alone.

The use of frequency based training was also studied in Chapter 4; training of networks was conducted for the same weight connection types used for step response training. It was found that for weights w_{64} and w_{46} – which were troublesome for step response training – the use of frequency based inputs greatly improved the generalization of the network. When a step input was modeled using the frequency based trained RGNNs, the error results for networks where w_{64} and w_{46} were trained fell below the maximum acceptable error. This trend did not continue for the remainder of weight

connection types and initial population intervals; the use of frequency based training in general created less generalized networks than using step response to train RGNNs with CAs.

Chapter 5 studied the use of a common non-gradient training algorithm, the GA. Based upon the findings during initial testing, the GA presented in Chapter 3 was modified to suit the needs of training a black box neural network model. To create a black box model the mutation algorithm was altered to ensure that any mutated weights were not limited to a certain interval. Also, initial training results showed that the GA had difficulties overcoming the accumulative error. It was found that if the GA was allowed to calculate new populations with no restrictions, the error of newly created populations would oscillate around a large error and would never improve. To eliminate this from occurring, a limitation was placed on the new population to ensure that the error of the new population was never worse than the error of the old population. If the error was larger, an individual from the old population would replace the newly calculated individual.

Unlike the training results using CA when one weight was trained at a time, for both step and frequency based training the GA proved completely ineffective as none of the trained networks fell below the maximum allowable error. Also unlike training results for the CA, when the number of weights trained increased from one to two the error of the trained networks decreased. This trend continued when the number of trained weights increased to include all required connections. When all possible weights in the network were trained using GAs the average minimum error was more than when only required weights were trained, but the error was still less than when one or two weights were trained at a time.

It was also found that like the CA, as the initial population size decreased, the average minimum error of networks trained using GAs also decreased. The training of RGNNs using GAs showed that step response training provided more generalized results when compared to frequency based training; this is not to say that the results themselves

were acceptable, just more generalized than frequency based training results. This was the same trend noticed during the training of RGNNs using CAs.

6.2 Conclusions

The prime objective of this particular study was to investigate the competency of using RGNNs along with the CA training method; to do so a simple stable linear system was used. This study was conducted by training networks using step response and frequency based training with a variety of initial population intervals and by varying the number of weights being trained in the RGNN. The prime objective was met and based upon the results; the first conclusion is as follows: the robustness of training RGNNs using the CA is dependent upon the initial population of weights. The characteristics which were noticed to affect the initial population's robustness included the population interval size and the number of weights being optimized as mentioned during the discussion of results.

A second objective was to complete a comparative study of CAs to the GA, which is also a non-gradient training method. Two modifications were made to the GA; however, only one showed a noticeable affect on the training of RGNNs using GAs. This modification was the assurance that the error of the new population never increased when compared to the previous population. If mating occurred and either child from the new population had a higher error than either of its parent from the old population, then the parent replaced the child in the new population. The limitation of the new population led to populations converging on an incorrect set of weights quickly and not moving towards an acceptable set after doing so. This phenomenon was not as noticeable with the CA because it has a mechanism to allow the weights to shift in a specific direction rather then randomly, increasing the chances of reaching the global minimum error if the collapsing of the population occurs at the incorrect points.

Based on this observation, the following conclusion is as forwarded: when using GAs, a specific algorithm must be chosen which will allow the calculation of new

population weights to move freely but at the same time ensure a stable output from the RGNN. Comparing the outputs for the GA and the CA, it was found that the CA consistently created networks for both step response and frequency based training with a significantly lower error for the same type of inputs. It was also concluded that the CA will produce more generalized RGNNs than the GA.

It is concluded that based upon the results of training RGNNs using the CA and GA when step response and frequency based training data sets were used, networks trained using step response will tend to be more generalized in the majority of cases. One observation which should be noted was that for step response training using CAs, weights w_{46} and w_{64} could not achieve the maximum allowable error when only one weight was optimized. However, when frequency based training data was used a significantly more generalized RGNN was obtained when only w_{46} and w_{64} were optimized. Further studies as to why these specific weights proved to be troublesome must be further explored in order to create a valid RGNN using the CA and achieve the global objective.

As stated in Chapter 1 and expanded upon in Chapter 4, the network studied was set to a predetermined number of neurons based upon the order and characteristics of the test model. The focus of the studies performed was to complete a sensitivity study of RGNNs trained using the CA. The sensitivity study performed did not include effects based on changing the number of neurons. The limitation upon the number of neurons (because the number of neurons used created an exact representation) creates a "grey box". Therefore, it is also concluded that the results obtained do not negate or validate the use of RGNNs using the CA for modeling load sensing pumps. To validate the results a true "black box" model must be created, and this was not completed during this investigation.

6.3 Recommended Future works

The intent of using two different non-gradient training algorithms was to study their use with RGNNs. As it was found that both training algorithms proved ineffective

for training RGNNs at large initial population intervals, a study of RGNNs should be conducted which considers other training algorithms. One such algorithm which should be considered is the use of Back Propagation (BP). As stated earlier, BP was not considered because the focus of the thesis was the use of non-gradient training methods as they are predisposed to the training of recurrent type networks. However, BP has been shown to be useful in the training of many network types. One disadvantage of non-gradient networks is the training of networks by comparing only the output error of each individual rather then the comparison of each neurons output. BP considers the output of each neuron in considering whether to increase or decrease the value of neural weight. This would increase the chance of a weight finding a global optima rather than an individual's local minimum. And in turn, this may help find the entire networks global optima.

The use of non-gradient methods should not be ignored in further studies based on the findings discussed in this thesis. Chapter 2 described the fundamental improvements of non-gradient methods in comparison to gradient methods such as BP. Each training algorithm can be modified in a number of methods; one such modification was considered for the use of CAs. A modified reflection point method was used to alleviate collapsing of the complex in a local rather than global minimum. Therefore it is suggested that the CA should be modified to incorporate the superior nuances of the GA.

It was found that the GA did not produce the same results as the CA. However, the GA uses one mechanism during mating which could prove advantageous to the CA, and may in the process decrease the training time. The GA uses the fitness factor of an individual to calculate the probability that one of its weights will be chosen for mating. A similar mechanism should be incorporated into the CA. When the calculation of the centroid is conducted, rather than each individual contributing equally the error of each individual should be used to evaluate how much it will contribute to the centroid. The lower the error an individual has, the closer the centroid will be to that individual.

Another possible improvement which can be made is the incorporation of BP tactics. BP modifies the weights of each neuron individually rather then as a group. Similarly, if the desired output of each neuron is known then rather than producing rankings based on the output of the individual, the rankings can be made based upon weight output. Two methods could be used for finding the output of each neuron. If the structure of the model is known then the output of each neuron can be discovered; this is similar to the exact model produced for the transfer function considered in the previous studies. However, this would create a grey box model rather then a black box model; this would be contrary to the desired creation of a black box model. Another method would be to use BP to predict the output of each neuron. Chapter 3 showed the difficulty in creating the gradients needed for such predictions; however, it is feasible to create such predictions. In doing so, a training algorithm would be made which would apply either CAs or GAs along with the BP neuron output prediction.

For the investigation conducted the number of neurons used for creating models using RGNNs was held constant. Therefore, it is recommended that further studies be conducted which investigate how the number of neurons used in an RGNN can affect the robustness of the network when trained using CAs. Once the number of neurons used for an RGNN is tested, it is recommended that the study of RGNNs trained using the CA be considered further to meet the global objective of creating a non-linear dynamic model for a load sensing pump using neural networks.

References

Andersson, J. (2001). *Multiobjective Optimization in Engineering Design – Applications to Fluid Power Systems*. Ph. D. Thesis, Division of Fluid and Mechanical Engineering Systems, Department of Mechanical Engineering, Linkoping University, Linkoping, Sweden.

Bitner, D. (1986). *Analytical and Experimental Analysis of a Load Sensing Pump*. M. Sc. Thesis, Department of Mechanical Engineering, University of Saskatchewan, Saskatoon, Saskatchewan.

Burton, R.T., Sargent, C.M., Schoenau, G.J., Anderson, D. (1993). The use of multiple independent gains for a repetitive low frequency duty cycle in a hydraulic system. *Proceedings of the Second Japan Hydraulic and Pneumatic Society International Symposium on Fluid Power*.

Burton, R.T., Ukrainetz, P.R., Nikiforuk, P. N., G.J.Schoenau. (1999). Neural Networks and Hydraulic Control – From Simple to Complex Applications. *Proceedings of the Institute of Mechanical Engineers*, Vol. 213, Part 1, Pp. 349-358.

Darwin, C. (1859). *On the Origin of Species*. New York Universal Press, Washington Square, NY.

Goldberg, D.E. (1989). Genetic *Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Publishing Company, Reading, MA.

Goulermas, J.Y., Liatsis, P., Zeng, X.J., Cook, P. (2007). Density-Driven Generalized Regression Neural Networks (DD-GRNN) for Function Approximation. *IEEE Transactions on Neural Networks*, Vol. 18, No. 6, Pp. 1683-1696.

Gupta, M., Jin, L., Homma, N. (2003). *Static and Dynamic Neural Networks – From Fundamentals to Advanged Theory*. John Wiley & Sons Inc., Hoboken, NJ.

Haykin, S. (1999). *Neural Networks – A Comprehensive Foundation*. Prentice Hall, Upper Saddle River, NJ.

Krus, P. (1988). *On Load Sensing Fluid Power Systems*. Ph. D. Thesis, Division of Fluid and Mechanical Engineering Systems, Department of Mechanical Engineering, Linkoping University, Linkoping, Sweden.

Lamontagne, D. (2001). *Investigations in Modeling a Load Sensing Pump Using Neural Networks*. M. Sc. Thesis, Department of Mechanical Engineering, University of Saskatchewan, Saskatoon, Saskatchewan.

Li, Y. (2007). *Investigation in Modeling a Load-Sensing Pump Using Dynamic Neural Unit Based Dynamic Neural Networks*. M. Sc. Thesis, Department of Mechanical Engineering, University of Saskatchewan, Saskatoon, Saskatchewan.

Mahanti, G.K., Chakraborty, A., Das, S. (2005). Floating-point Genetic Algorithm for design of a Reconfigurable Antenna Arrays by phase-only control. *Asia-Pacific Microwave Conference Proceedings*, Vol. 5, (4 pages).

McNamara, J.M., Edge, K.A., Vaughan, N.D. (1997). Hybrid Analytical/Neural Network Model of Variable Displacement Pump Dynamics. *Fluid Power Systems and Technology: Collected Papers*, FPST-Vol. 4/DSC-Vol. 63, Pp. 71-76.

Merritt, H. (1967). *Hydraulic Control Systems*. John Wiley & Sons, New York, NY.

Pollmeier, K., Burrows, C.R., Edge, K.A. (2004). Condition Monitoring of an Electrohydraulic Position Control System Using Artificial Neural Networks.

ASME International Mechanical Engineering Congress and Exposition, November 13-20 2004, Pp. 137-146.

Qian, W. (1998). *Neural network control of non-linear hydraulic system*. M. Sc. Thesis, Department of Mechanical Engineering, University of Saskatchewan, Saskatoon, Saskatchewan.

Song, Y.H., Johns, A.T. (1998). Application of Fuzzy Logic in Power Systems: Part 2 Comparison and integration with expert systems, neural networks and genetic algorithms. *Power Engineering Journal*, August 1998, Pp. 185-190.

Srivastava, V (1998). *Neural-Control of Unknown Dynamic Systems*. M. Sc. Thesis, Department of Mechanical Engineering, University of Saskatchewan, Saskatoon, Saskatchewan.

Werbos, P.J. (1990). Backpropagation Through Time: What It Does and How to Do It. *Proceedings of the IEEE*, No. 10, Vol. 78, Pp. 1550-1560.

Wiens, T. (2008a). *Online Learning of a Neural Fuel Control System for Gaseous Fueled SI Engines*. Ph. D. Thesis, Department of Mechanical Engineering, University of Saskatchewan, Saskatoon, Saskatchewan.

Weins, T., Burton, R.T., Schoenau, G.J. and Bitner, D. (2008). Recursive Generalized Neural Networks (EGNN) for the Modeling of a Load Sensing Pump., Bath-ASME Joint Conference on Fluid Power, Transmission and Control, Bath, England, Sept 4-7, (8 pages) (CD Rom).

Wu, D. (2003). *Modeling and Experimental Evaluation of a Load-Sensing and Pressure Compensated Hydraulic System*. Ph. D. Thesis, Department of Mechanical Engineering, University of Saskatchewan, Saskatoon, Saskatchewan.

Xu, X.P. (1997). *Experimental Modeling of a Hydraulic Load Sensing Pump Using Neural Networks*. Ph. D. Thesis, Department of Mechanical Engineering, University of Saskatchewan, Saskatoon, Saskatchewan.

Yu, F., Gupta, N., Hoy, J. (2005). Non-Intrusive Pressure Measurement Based on Ultrasonic Waves. *Insight May 2005*, No. 5, Vol. 47, Pp. 285-288.

Zhang, H., Ukrainetz, P.R., Nikiforuk, P.N., Burton, R.T. (1996). Implementation of Neural Network Approach in MIMO Electrohydraulic Servosystems Control. *Proceedings of the International Conference on Control*, University of Exeter, September 1996, Pp. 1468-1473.

Appendix A: Derivation of Exact Representation for Recurrent Generalized Neural Network (RGNN)

The following discussion is an expansion of the exact representation of a RGNN for a transfer function given in Chapter 2. The transfer function considered is,

$$\frac{Y(s)}{X(s)} = \frac{s+4}{s^2+2s+9}. \tag{A.1}$$

Equation A.1 describes a single-input single-output (SISO) system which contains a zero at,

$$s = -4$$

and two poles at,

$$s = -1 \pm i2\sqrt{2}\,.$$

Because both poles lie on the left hand plane, for a step input a stable output is achievable. Taking the inverse Laplace transform of Equation 2.4 gives,

$$\frac{d^2y}{dt^2} + 2\frac{dy}{dt} + 9y = \frac{dx}{dt} + 4x\,. \tag{A.2}$$

If it is assumed that $\Delta t \to 0$, then the differential components in A.2 can be written as,

$$\frac{dy}{dt} = \frac{y^k - y^{k-1}}{\Delta t}, \tag{A.3}$$

$$\frac{d^2y}{dt^2} = \frac{y^k - 2y^{k-1} + y^{k-1}}{\Delta t^2}. \tag{A.4}$$

Therefore using past time step information from both input and output, A.2 can be simplified to,

$$\frac{y^k - 2y^{k-1} + y^{k-2}}{\Delta t^2} + \frac{2y^k - 2y^{k-1}}{\Delta t} + 9y^k = \frac{x^k - x^{k-1}}{\Delta t} + 4x^k\,. \tag{A.5}$$

Collecting like terms for y^k and x^k yields,

$$y^k\left[\frac{1}{\Delta t^2} + \frac{2}{\Delta t} + 9\right] + y^{k-1}\left[-\frac{2}{\Delta t^2} - \frac{2}{\Delta t}\right] + y^{k-2}\left[\frac{1}{\Delta t^2}\right] = x^k\left[\frac{1}{\Delta t} + 4\right] + x^{k-1}\left[-\frac{1}{\Delta t}\right]. \tag{A.6}$$

$$Ay^k + By^{k-1} + Cy^{k-2} = Dx^k + Ex^{k-1}$$
$$y^k = -\frac{B}{A}y^{k-1} - \frac{C}{A}y^{k-2} + \frac{D}{A}x^k + \frac{E}{A}x^{k-1}. \quad \textbf{(A.7)}$$

In order to find the coefficients for Equation A.7 the RGNN must be formed and examined. Equation A.7 shows a two step time delay for the output and a single delay for the input. Based on the number of delay paths needed, equation A.7 can be represented using the RGNN schematic in Figure A.1 where all return paths have a one time step delay.

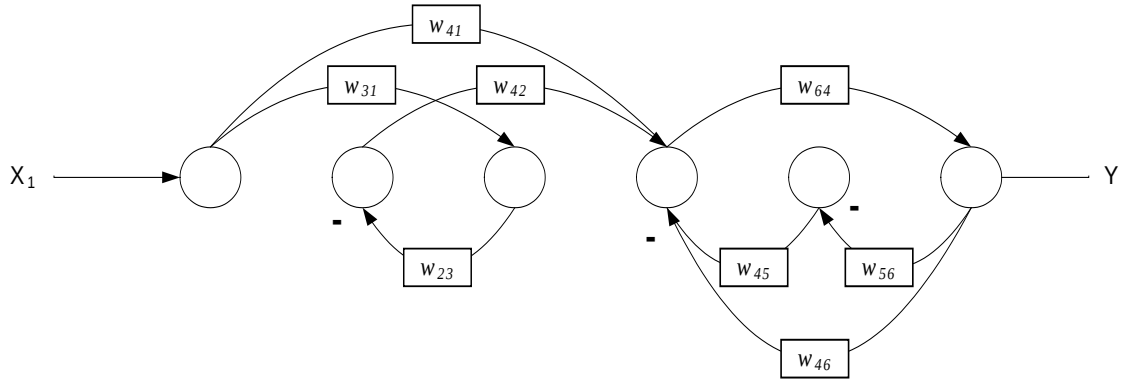


Figure A.1: Schematic of exact representation of Equation A.7 using a RGNN.

Comparing Figure A.1 to Equation A.7 the values for the RGNN weights can be determined using values for the coefficients found above,

$$-\frac{B}{A} = (-w_{46}) \rightarrow w_{46} = \frac{B}{A}$$

$$-\frac{C}{A} = (-w_{45})(-w_{56}) \rightarrow w_{45} = \frac{C}{A}, w_{56} = -1$$

$$\frac{D}{A} = w_{41} \rightarrow w_{41} = \frac{D}{A}$$

$$\frac{E}{A} = w_{31}(-w_{23})w_{42} \rightarrow w_{31} = 1, w_{23} = -1, w_{42} = \frac{E}{A}$$

For a SISO system, RGNNs are described by,

$$Y^k = W_S X^k - W_D Y^{k-1}, \tag{A.8}$$

where W_S and W_D are the weight matrices for static and dynamic connections which are upper and lower triangular matrices respectively. The overall weight matrix is given by,

$$W = W_S + W_D. \tag{A.9}$$

And for the RGNN exact representation being considered,

$$W = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & w_{23} & 0 & 0 & 0 \\ w_{31} & 0 & 0 & 0 & 0 & 0 \\ w_{41} & w_{42} & 0 & 0 & w_{45} & w_{46} \\ 0 & 0 & 0 & 0 & 0 & w_{56} \\ 0 & 0 & 0 & w_{64} & 0 & 0 \end{bmatrix}. \tag{A.10}$$

If a time step of $\Delta t = 0.02$ seconds is considered the values for the coefficients in A.7 are,

$$A = \frac{1}{\Delta t^2} + \frac{2}{\Delta t} + 9 = \frac{1}{(0.02)^2} + \frac{2}{0.02} + 9 = 2609$$

$$B = -\frac{2}{\Delta t^2} - \frac{2}{\Delta t} = -\frac{2}{(0.02)^2} - \frac{2}{0.02} = -5100$$

$$C = \frac{1}{\Delta t^2} = \frac{1}{(0.02)^2} = 2500$$

$$D = \frac{1}{\Delta t} + 4 = \frac{1}{0.02} + 4 = 54$$

$$E = -\frac{1}{\Delta t} = -\frac{1}{0.02} = -50$$

Using the values for the coefficients the values for the neural network weight matrix, *W*, in equation A.10 becomes,

$$W = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0.0207 & -0.0192 & 0 & 0 & 0.9582 & -1.9547 \\ 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}.$$

Figure A.2 shows the output of the RGNN in comparison to the continuous output for the transfer function in Equation A.1. The RGNN was created using two sample times, 0.005 seconds and 0.02 seconds. It can be seen in Figure A.2 that as the sample time increases, the accuracy of the RGNN decreases.

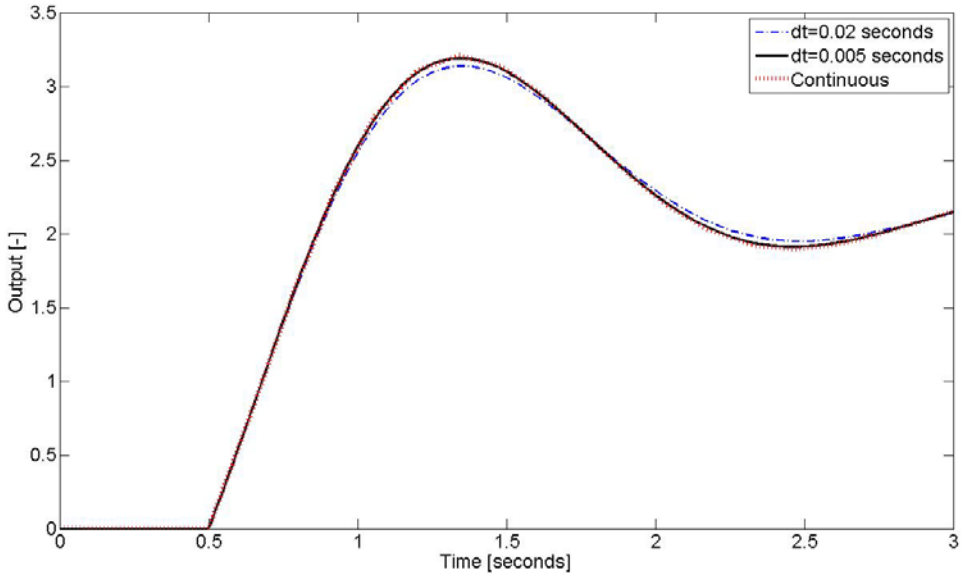


Figure A.2: Output comparison of RGNN at 0.02s and 0.005s compared to the continuous output of the transfer function in Equation A.1.

Appendix B: Calculation of Updated Neural Network Weights Using Non-Gradient Methods.

B.1: Introduction

The following appendix will outline the sample calculations necessary to complete an iteration of the training process for both the complex algorithm (CA) and genetic algorithm (GA). The calculations will be completed for a static feed-forward neural network (FFNN) which was considered in Chapter 3; a schematic for the network is shown in Figure B.1.

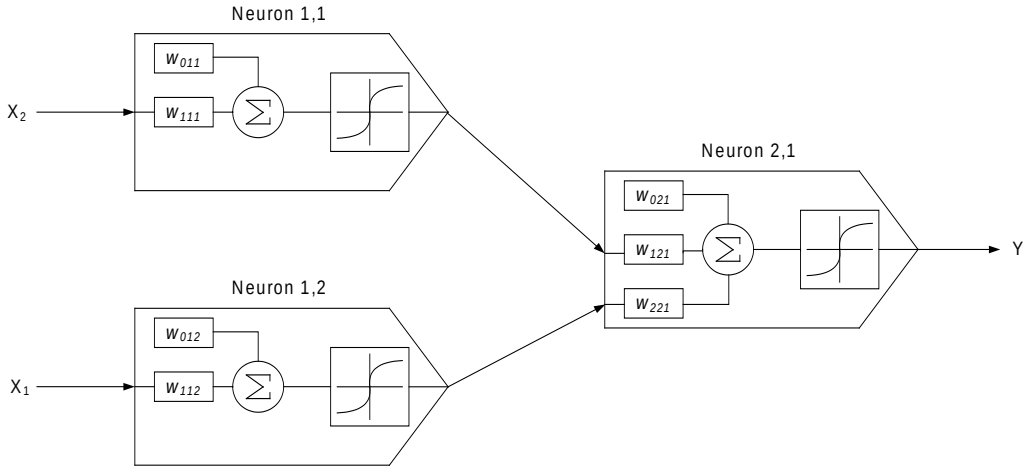


Figure B.1: Schematic of FFNN considered for outline of training algorithms.

The weighting matrix for Figure B.1 is written as,

$$W = \begin{bmatrix} w_{011} & w_{111} & w_{012} & w_{112} & w_{021} & w_{121} & w_{221} \end{bmatrix}, \qquad \textbf{(B.1)}$$

The FFNN will be trained to model the mathematical equation,

$$y = x + x^{1/2} + 2, \qquad \textbf{(B.2)}$$

where y is the output of the neural network and x is the input. Batch training will be used; for batch training the error considered is the least squared error given by,

$$E(k) = \frac{1}{2}\frac{1}{N}\sum_{k=1}^{N} e(k)^2 , \qquad \textbf{(B.3)}$$

where k is the time step being considered, N is the total number of time steps in the batch and $e(k)$ is the difference between the output of the neural network and the desired output of the neural network. For both CA and GA the initial population is given in Table B.1 along with each individuals corresponding error.

Table B.1: Initial population of weights and corresponding error.

Individual	w_{011}	w_{111}	w_{012}	w_{112}	w_{021}	w_{121}	w_{221}	E_i
1	0.8382	0.8292	0.7142	0.7912	0.4799	0.0934	0.9917	45440.20
2	0.3902	0.4406	0.0066	0.7777	0.9006	0.7098	0.5805	42185.88
3	0.9181	0.7549	0.5861	0.8833	0.0265	0.4554	0.9461	46186.07
4	0.1559	0.9671	0.5759	0.4114	0.1029	0.7538	0.2423	48011.48
5	0.1680	0.7507	0.9734	0.2773	0.3415	0.4124	0.7218	45950.36
6	0.4612	0.4823	0.9062	0.0180	0.9137	0.8917	0.0713	43871.46
7	0.1093	0.0980	0.7658	0.9466	0.5638	0.4087	0.5459	46070.98
8	0.7010	0.2208	0.0387	0.8139	0.7986	0.4011	0.4596	44972.19

B.2: Calculations for Complex Algorithm (CA)

The first step in calculating an updated set of weights using CA is to identify the individual with the highest error. By inspecting the error for each individual in Table B.1 it can be seen that individual 4 has the highest error (W^h); the following calculations will outline how to create a new individual to replace W^h.

The second step is to calculate a centroid which consists of the average value of each individual for a specific weight except W^h; the weight to be considered for the following calculations is w_{011}. The centroid is given by,

$$\overline{W} = \frac{1}{n-1}\sum_{i=1}^{n} W^i , \quad where\ W^i \neq W^h , \qquad \textbf{(B.4)}$$

therefore,

$$\overline{W} = \frac{1}{6-1}\left(0.8382 + 0.3902 + 0.9181 + 0.1680 + 0.4612 + 0.1093 + 0.7010\right) = \frac{1}{5}\left(2.5614\right)$$

$$\overline{W} = 0.5123\,.$$

The third step is the calculation of the new reflection point given by,

$$W^r = \overline{W} + \alpha\left(\overline{W} - W^h\right), \qquad \textbf{(B.5)}$$

where α is the reflection coefficient. For our purposes the value for α will be 1.3 as suggested by Andersson [2001]. Therefore,

$$W^r = 0.5123 + 1.3\left(0.5123 - 0.1559\right)$$

$$W^r = 0.9755\,.$$

The new error for the output produced by the FFNN with W^r is 41693.4; which is less then the error for W^h.

If for instance the error produced by W^r was greater then W^h then a modified reflection algorithm [Andersson, 2001] is given by,

$$W^{r(new)} = \left[W^{r(old)} + \varepsilon\overline{W} + \left(1-\varepsilon\right)W^l\right]/2 + \left(\overline{W} - W^l\right)\left(1-\varepsilon\right)\left(2R-1\right), \qquad \textbf{(B.6)}$$

where,

$$\varepsilon = \left(\frac{n_r}{n_r + k_r - 1}\right)^{\frac{n_r + k_r - 1}{n^r}}. \qquad \textbf{(B.7)}$$

In Equations B.6 and B.7 W^l is the individual with the lower error, R is a random value on the interval [0, 1], n_r is a constant chosen to be $n_r = 4$, and k_r is the number of times the modified reflection algorithm has been attempted at a given generation. For w_{011} the modified reflection algorithm would produce,

$$\varepsilon = \left(\frac{4}{4+1-1}\right)^{\frac{4+1-1}{4}} = 1\,.$$

Therefore if $R = 0.9501$,

$$W^{r(new)} = [0.9755 + (1)(0.5123) + (1-1)0.3902]/2 + (0.5123 - 0.3902)(1-1)(2(0.9501)-1)$$
$$W^{r(new)} = 0.7439$$
$$E^{r(new)} = 42915.5\,.$$

B.3: Calculations for Genetic Algorithm (GA)

The first step in training a neural network using GA is to find the total error of the population,

$$E_P = \sum_{i=1}^{n} E^{(i)}\,, \qquad \textbf{(B.8)}$$

$$E_P = 45440.2 + 42185.9 + 46186.1 + 48011.5 + 45950 + 4 + 43873.5 + 46070.9 + 44972.2$$
$$E_P = 362688.62\,.$$

Once the total error is found the fitness factor for each individual must be calculated. The fitness factor being considered behaves similar to a probability wheel; the chances of a specific weight being found is dependent on how good the individual is, or is our case how low the error is. The fitness for an individual is given by,

$$f^{(i)} = \frac{E_p - E^{(i)}}{E_p}\,. \qquad \textbf{(B.9)}$$

The fitness factor for individual 1 is,

$$f^{(1)} = \frac{E_p - E^{(1)}}{E_p} = \frac{362688.6 - 45440.2}{362688.6}$$
$$f^{(1)} = 0.8747\,,$$

with the fitness for all individuals given in Table B.2.

Table B.2: Values for individual fitness factors in initial population.

Individual	1	2	3	4	5	6	7	8	total
$f^{(i)}$	0.8747	0.8837	0.8727	0.8676	0.8733	0.8790	0.8730	0.8760	7.0000

The next step is to create a tentative population of weights which is based on the probability of a specific individual being chosen based on fitness factor. The methodology used to choose a new population stems from a function which is contained in MATLAB coding called *rand*; this function produces a random number on the interval [0, 1]. Because the total fitness of the population is 7.0000, a ratio of an individual's fitness factor compared to overall fitness of a population was considered. For instance, if individual 1 is considered,

$$f_{ratio}^{(i)} = \frac{f^{(1)}}{f_p} \qquad \textbf{(B.10)}$$

$$f_{ratio}^{(i)} = \frac{0.8747}{7.0000} = 0.1250\,.$$

Because rand will create a number of interval [0, 1] the fitness ratios for each individual must be placed in subsequent order to create interval ranges as shown in Table B.3. Consider a new population which is being formed; the first individual is obtaining its $w_{011}^{(new)}$. The rand function outputs a value 0.0798; this value corresponds to the fitness range of individual 1. Therefore for the new population, the first new individual at $w_{011}^{(new)}$ will be $w_{011}^{(old)} = 0.8382$.

Table B.3: Fitness ratios along with fitness ranges for population.

Individual	1	2	3	4	5	6	7	8
$f_{ratio}^{(i)}$	0.1250	0.1262	0.1247	0.1239	0.1248	0.1256	0.1247	0.1251
low	0.0000	0.1250	0.2512	0.3759	0.4998	0.6246	0.7501	0.8749
high	0.1250	0.2512	0.3759	0.4998	0.6246	0.7501	0.8749	1.0000

Once a tentative new population has been chosen, each individual must chose a mate to create offspring; the mates for each individual are chosen randomly. The tentative new population along with the chosen mates is shown in Table B.4.

Table B.4: Tentative new population with individuals corresponding mates.

Individual	w_{011}	w_{111}	w_{012}	w_{112}	w_{021}	w_{121}	w_{221}	Mate
1	0.8382	0.8292	0.5759	0.9466	0.7986	0.4554	0.5805	5
2	0.1093	0.4406	0.9062	0.8139	0.9137	0.4087	0.9917	6
3	0.3902	0.0980	0.5759	0.8139	0.9137	0.8917	0.9461	8
4	0.3902	0.4406	0.0066	0.4114	0.5638	0.7538	0.5459	7
5	0.1559	0.9671	0.5759	0.4114	0.7986	0.0934	0.9461	1
6	0.8382	0.4823	0.9062	0.8139	0.3415	0.7538	0.5805	2
7	0.4612	0.4823	0.5759	0.7912	0.1029	0.4087	0.5459	4
8	0.7010	0.0980	0.5861	0.2773	0.9137	0.7098	0.4596	3

Now that each of the individuals has found a mate the error of each individual must be found using Equation B.3. Mating involves two steps, heuristic crossover and non-uniform mutation. To complete the heuristic crossover process the error of each mate must be compared. Once mating occurs, the parent with the lowest error, I^A, and the highest error, I^B, are used to find two new offspring,

$$\begin{aligned} I^{B(new)} &= I^A + r\left(I^A - I^B\right), \\ I^{A(new)} &= I^A \end{aligned} \qquad \textbf{(B.11)}$$

where r is a random value between zero and one. The mate for individual 1 is 5 who have errors of 44002 and 44010 respectively, therefore for w_{011} with $r = 0.5$,

$$w_{011}{}^{5(new)} = w_{011}{}^{1} + r\left(w_{011}{}^{1} - w_{011}{}^{5}\right) = 0.8382 + 0.5\left(0.8382 - 0.1559\right)$$
$$w_{011}{}^{5(new)} = 1.1794\,,$$

and,

$$w_{011}{}^{1(new)} = w_{011}{}^{1} = 0.8382\,.$$

The final step in the mating process is non-uniform mutation; this can occur in a wide variety patterns ranging from one weight in every individual being changed at every generation to one weight in the entire population being changed at each generation.

Non-uniform mutation is governed by,

$$w^{i(mutate)} = w^i + \left(b - w^i\right) f\left(G\right) \qquad \text{If } r_1 < 0.5 \qquad \textbf{(B.12)}$$

$$w^{i(mutate)} = w^i - \left(a + w^i\right) f\left(G\right) \qquad \text{If } r_1 \geq 0.5 \qquad \textbf{(B.13)}$$

where,

$$f\left(G\right) = \left(r_2 \left(1 - \frac{G}{G_{\max}} \right) \right)^c \qquad \textbf{(B.14)}$$

where r_1 and r_2 are random numbers between zero and one, b and a are the upper and lower boundaries respectively of the possibly weight range given by the user. G is the number of times a new population has been calculated. This is also referred to as the number of generations; G_{max} is the maximum number of generations, and c is system parameter determining the degree of non-uniformity.

To illustrate mutation let us consider w_{011} in the new population with a boundary of [-5, 5], a non-uniformity parameter of $c = 3$, and r_1 and r_2 are 0.3 and 0.6 respectively. If the maximum number of generations is $G_{max} = 50$ and the current generation is $G = 10$ then,

$$f\left(G\right) = \left(0.6 \left(1 - \frac{10}{50} \right) \right)^3 = 0.1106,$$

and since $r_1 < 0.5$,

$$w_{011}^{(mutate)} = 0.8382 + \left(5 - 0.8382\right)\left(0.1106\right)$$

$$w_{011}^{(mutate)} = 1.2985.$$

Given a mutation process which only affects w_{011} in the first individual, the new population of individuals for the GA is given in Table B.5.

Table B.5: New population using heuristic crossover and non-uniform mutation.

Ind.	w_{011}	w_{111}	w_{012}	w_{112}	w_{021}	w_{121}	w_{221}
1	0.8382	0.8292	0.5759	0.9466	0.7986	0.4554	0.5805
2	0.1093	0.4406	0.9062	0.8139	0.9137	0.4087	0.9917
3	0.3902	0.0980	0.5759	0.8139	0.9137	0.8917	0.9461
4	0.3902	0.4406	0.0066	0.4114	0.5638	0.7538	0.5459
5	1.1794	0.7603	0.5759	1.2142	0.7986	0.6364	0.3978
6	-0.2552	0.4198	0.9062	0.8139	1.1998	0.2362	1.1973
7	0.3547	0.4198	-0.2780	0.2215	0.7943	0.9263	0.5459
8	0.2348	0.0980	0.5708	1.0823	0.9137	0.9826	1.1893

Appendix C: Simulation Code

The following code contains the MATLAB based code used to train RGNNs using both the CA and GA.

C.1 RGNN Code (`dynamic_rgnn.m`)

The purpose of the following program is to create the output of a recurrent generalized neural network of a SISO based RGNN.

Input(s):

- Individual's weight matrix in string formation (`W_string`)
- Number of neurons in network (`neurons`)
- Input to the network and desired output (`x, yd`)

Output(s):

- Output of RGNN (`y`)
- Error of RGNN (`E`)

```
function [y,E]=dynamic_rgnn(x,yd,neurons,W_string)

[W,B]=string2square(neurons,W_string);
Ws=tril(W,-1); %static matrix
Wd=triu(W); %dynamic matrix
time_step=size(x,2);
Y=zeros(neurons,time_step);
Y(1,:)=x;
%calculate outputs
for k=1 %static component, ie. input to input neuron
    for i=2:(neurons-1)
        Y(i,k)=Ws(i,:)*Y(:,k)+B(i); %for non-linear
tanh(Ws(i,:)*Y(:,k)+B(i));
    end
    for i=neurons
```

```
        Y(i,k)=Ws(i,:)*Y(:,k)+B(i); %use linear output for output
neuron
    end
end
for k=2:time_step %at the moment this is static
    for i=2:(neurons-1)
        Y(i,k)=Ws(i,:)*Y(:,k)-Wd(i,:)*Y(:,k-1)+B(i);
        %for non-linear =tanh(Ws(i,:)*Y(:,k)+B(i));
    end
    for i=neurons
        Y(i,k)=Ws(i,:)*Y(:,k)+B(i);
        %use linear output for output neuron, also there is no dynamic
to output neuron Wd=0
    end
end
y=Y(neurons,:); %output of network

%% Calculate the error
e2=zeros(1,size(x,2));
for i=1:size(x,2)
    e2(i)=(y(1,i)-yd(1,i))^2;
end
E=[sum(0.5*e2) 0]; %second entry is to define when infinit error is
reached
if isnan(E(1,1))==1
    i=1;
    while i<=time_step&&E(1,2)==0
        if y(i)==inf||isnan(y(i))==1
            E(1,2)=i;
        end
        i=i+1;
    end
elseif E(1,1)==inf
    E(1,2)=time_step+1;
end
if isnan(E(1,1))==1
    E(1,1)=inf;
end
```

```
if isinf(E(1,1))~=1
    E(1,2)=time_step+2;
end
```

C.1.1 Changing Population Row Vector to a Square Matrix (`string2square.m`)

The purpose of the following code is to create a square matrix which is used for linear algebra calculation of `dynamic_rgnn.m`. The matrix is produced as a row vector to simplify the calculation of the new population during training.

Input(s):

Number of neurons

- Individual represented as a row vector (`W_string`)
- Number of neurons in the network (`neurons`)

Output(s):

- Individual represented as a square matrix (`W_square, B`)

```
function [W_square,B]=string2square(neurons,W_string)

W_square(1,1:neurons)=W_string(1,1:neurons);
for j=1:(neurons-1)

W_square(j+1,1:neurons)=W_string(1,(j*neurons+1):(j*neurons+neurons));
end

B=W_string((neurons*neurons+1):(neurons*neurons+neurons));
```

C.2 Complex Algorithm (`complex_rgnn_august26_2010.m`)

The purpose of the following code is to complete training of an RGNN using the CA.

```
clear all
tic
% specify network parameters
neurons=6;
pop=neurons*neurons+10;%should be an even number to correspond with
genetic
max_count=50000; %number of times a new point can be calculated
max_kr=1500; %number of times a single point loop can be calculated
max_time=9000; %maximum time allowed for training in seconds
data_out=zeros(7,max_count); %preset output data set
%% set training parameters
high=5;
low=-5;
% load training data
load('dynamic_3step'); % Load initial data

% initiate intervals for weights
W_small=zeros(neurons,neurons,pop);
B_small=zeros(1,neurons,pop);
if rem(high,1)==0
    for i=1:pop
        W_big(:,:,i)=randint(neurons,neurons,[low,high]);
        B_big(:,:,i)=randint(1,neurons,[low,high]);
        for j=1:neurons
            for k=1:neurons
                W_small(j,k,i)=rand(1)*(-1)^randint;
                B_small(1,k,i)=rand(1)*(-1)^randint;
            end
        end
    end
else
    W_big=zeros(neurons,neurons,pop);
```

```
    B_big=zeros(1,neurons,pop);
    for i=1:pop
        for j=1:neurons
            for k=1:neurons
                W_small(j,k,i)=0.5*rand(1)*(-1)^randint;
                B_small(1,k,i)=0.5*rand(1)*(-1)^randint;
            end
        end
    end
end
W_interval=W_small+W_big;
W_interval(1,:,:)=0;
W_interval(neurons,neurons,:)=0;
% initial intervals for bias
B_init=B_small+B_big; %B_init=zeros(1,neurons,pop) if not ALL weights
are calc.
B=B_init; %save initial information

% create initial weight matrix
W_exact=[0,0,0,0,0,0;
    0,0,-1,0,0,0;
    1,0,0,0,0,0;
    0.020697585281717,-0.019164430816405,0,0,0.958221540820238,-
1.954771943273285;
    0,0,0,0,0,-1;
    0,0,0,1,0,0;];
for i=1:pop
    W_square(:,:,i)=W_exact+W_interval(:,:,i);
end

for i=1:pop
    [W(i,:)]=square2string(neurons,W_square(:,:,i),B(:,:,i));
end
W_init=W; %save initial information
%% save('Winit_complex_step_5to5_1','W_init')
% calculate output and weights for initial weight matrices
for i=1:pop
    [y(i,:),E(i,:)]=dynamic_rgnn(x,yd,neurons,W(i,:));
```

```
end
y_init=y; %save initial information
E_init=E; %save initial information

% find best and worst individuals
[ind,ind_h,ind_l]=max_min(E,pop);
w_h=W(ind_h(1,1),:);
w_l=W(ind_l(1,1),:);
count=1;
kr=1;
%% Calculate Complex
while
(mean(E(:,1)>0.02)||isnan(ind_h(1,2))==1)&&count<max_count&&kr<max_kr&&
toc<max_time
    %% creating centroid
    ind_cen=zeros(pop,1);
    for i=1:size(W,2)
        for j=1:pop
            if ind_h(1,1)~=ind(j,1)
                ind_cen(j,i)=W(j,i);
            end
        end
    end
    w_cen=zeros(1,size(W,2));
    for i=1:size(W,2)
        w_cen(i)=sum(ind_cen(:,i))/(pop-1);
    end
    %% reflection point
    w_ref=zeros(1,size(W,2));
    for i=1:size(w_ref,2)
        w_ref(i)=w_cen(i)+1.3*(w_cen(i)-w_h(i));
    end
    [y_ref,E_ref]=dynamic_rgnn(x,yd,neurons,w_ref);
    kr=1;
    if
(E_ref(1,1)<inf&&isnan(E_ref(1,1))==0)&&(ind_h(1,2)<inf&&isnan(ind_h(1,
2))==0)
        while E_ref(1,1)>ind_h(1,2)&&kr<max_kr
```

```
            nr=4;
            eps=(nr/(nr+kr-1))^((nr+kr-1)/nr);
            R=rand(1);
            w_ref2=zeros(1,size(W,2));
            for i=1:size(W,2)
                w_ref2(i)=(w_ref(i)+eps*w_cen(i)+(1-
eps)*w_l(i))/2+(w_cen(i)-w_l(i))*(1-eps)*(2*R-1);
            end
            [y_ref2,E_ref2]=dynamic_rgnn(x,yd,neurons,w_ref2);
            w_ref=w_ref2;
            y_ref=y_ref2;
            E_ref=E_ref2;
            kr=kr+1;
        end
    elseif
(E_ref(1,1)==inf||isnan(E_ref(1,1))==1)&&(ind_h(1,2)==inf||isnan(ind_h(
1,2))==1)
        while E_ref(1,2)<=ind_h(1,3)&&E_ref(1,2)~=0&&kr<max_kr
            nr=4;
            eps=(nr/(nr+kr-1))^((nr+kr-1)/nr);
            R=rand(1);
            w_ref2=zeros(1,size(W,2));
            for i=1:size(W,2)
                w_ref2(i)=(w_ref(i)+eps*w_cen(i)+(1-
eps)*w_l(i))/2+(w_cen(i)-w_l(i))*(1-eps)*(2*R-1);
            end
            [y_ref2,E_ref2]=dynamic_rgnn(x,yd,neurons,w_ref2);
            w_ref=w_ref2;
            y_ref=y_ref2;
            E_ref=E_ref2;
            kr=kr+1;
            if rem(kr,10)==0
                count
                kr
                min(E)
                max(E)
            end
        end
```

```
    elseif
(E_ref(1,1)==inf||isnan(E_ref(1,1))==1)&&(ind_h(1,2)~=inf||isnan(ind_h(
1,2))==0)
        while
(E_ref(1,1)>ind_h(1,2)||(E_ref(1,1)==inf||isnan(E_ref(1,1))==1))&&kr<ma
x_kr
            nr=4;
            eps=(nr/(nr+kr-1))^((nr+kr-1)/nr);
            R=rand(1);
            w_ref2=zeros(1,size(W,2));
            for i=1:size(W,2)
                w_ref2(i)=(w_ref(i)+eps*w_cen(i)+(1-
eps)*w_l(i))/2+(w_cen(i)-w_l(i))*(1-eps)*(2*R-1);
            end
            [y_ref2,E_ref2]=dynamic_rgnn(x,yd,neurons,w_ref2);
            w_ref=w_ref2;
            y_ref=y_ref2;
            E_ref=E_ref2;
            kr=kr+1;
        end
    end
    %% return to population
    y(ind_h(1,1),:)=y_ref;
    W(ind_h(1,1),:)=w_ref;
    E(ind_h(1,1),:)=E_ref;
    %perform new search for best and worst
    [ind,ind_h,ind_l]=max_min(E,pop);
    w_h=W(ind_h(1,1),:);
    w_l=W(ind_l(1,1),:);

data_out(:,count)=[count;ind_h(1,2);ind_h(1,3);ind_l(1,2);ind_l(1,3);kr
;toc];
    count=count+1;
    if rem(count,500)==0
        count
        ind_l
        ind_h
    end
```

```
end
ind=[[1:pop]' E];
ind_min=ind(1,:);
for i=2:pop
    if ind(i,2)<ind_min(1,2)
        ind_min(1,:)=ind(i,:);
    end
end
W_min=W(ind_min(1,1),:);
y_min=y(ind_min(1,1),:);
training_time=toc;
min(E),count,kr
save('complex_step_all_5to5_1')
```

C.2.1 Weight Matrix Transformation (`square2string.m`)

The purpose of the following program is to turn a square matrix representing an individual into a row vector representing the same individual.

Input(s):

- Number of neurons (`neurons`)
- Individual represented as a square matrix (`W_square`)
- Neuron Bias' (`B`)

Output(s):

- Individual represented as a row vector (`W_string`)

```
function [W_string]=square2string(neurons,W_square,B)

%Matrix Weights
for j=1:neurons
    W_string(1,((j-1)*neurons+1):((j-
1)*neurons)+neurons)=W_square(j,1:neurons);
end
%Neuron Bias'
W_string(1,(neurons*neurons+1):(neurons*neurons+neurons))=B;
```

C.2.2 Obtaining Maximum and Minimum Error (`max_min.m`)

The purpose of the following code is to assign each individual an index number, and then search for the maximum and minimum error values. Also, the corresponding individuals' locations are saved.

Input(s):

- Error Matrix (`E`)
- Number of individuals in the population (`pop`)

Output(s):

- Individual Matrix (which includes all indexed individuals, even if they are not the max or min values) (`ind`)
- Individual with the highest error (`ind_h`)
- Individual with the lowest error (`ind_l`)

```
function [ind,ind_h,ind_l]=max_min(E,pop)

ind=[[1:pop]' E];
ind_h=ind(1,:);

for i=2:pop
    if
(ind_h(1,2)==inf||isnan(ind_h(1,2))==1)&&(ind(i,2)==inf||isnan(ind(i,2)
)==1)
        if ind_h(1,3)>ind(i,3)
            ind_h=ind(i,:);
        end
    elseif
(ind_h(1,2)~=inf&&isnan(ind_h(1,2)==0))&&(ind(i,2)==inf||isnan(ind(i,2)
)==1)
        ind_h=ind(i,:);
    elseif ind_h(1,2)<inf&&ind(i,2)<inf
        if ind_h(1,2)<ind(i,2)
            ind_h=ind(i,:);
        end
    elseif ind_h(1,2)<inf&&(ind(i,2)==inf||isnan(ind(i,2))==1)
```

```
        ind_h=ind(i,:);
    end
end

ind_l=ind(1,:);
for i=2:pop
    if
(ind_l(1,2)==inf||isnan(ind_l(1,2))==1)&&(ind(i,2)==inf||isnan(ind(i,2)
)==1)
        if ind_l(1,3)<ind(i,3)
            ind_l=ind(i,:);
        end
    elseif
(ind_l(1,2)==inf||isnan(ind_l(1,2))==1)&&(ind(i,2)~=inf||isnan(ind(i,2)
)==0)
        ind_l=ind(i,:);
    elseif ind_l(1,2)<inf&&ind(i,2)<inf
        if ind_l(1,2)>ind(i,2)
            ind_l=ind(i,:);
        end
    elseif (ind_l(1,2)==inf||isnan(ind_l(1,2))==1)&&ind(i,2)<inf
        ind_l=ind(i,:);
    end
end
```

C.3 Genetic Algorithm (`genetic_rgnn_oct6_2010.m`)

The purpose of the following algorithm is to train a RGNN using the GA.

```
clear all
tic
% network data
neurons=6;
pop=neurons*neurons+10;
if rem(pop,2)==1
    pop=pop+1;
```

```
end
max_count=5000;
max_time=2.5*3600;
data_out=zeros(6,max_count);
data_E=zeros(pop,2*max_count);
load('dynamic_3step');
upper=5;
lower=-5;
%% Mutation Criteria
change_weight=(neurons+1):1:(neurons*neurons-1); %for all
%change_weight=[9 13 19 20 23 24 30 34]; % for exact
%change_weight=[9 13]; % for two weights
%change_weight=19; %for single weight
w_change=fix(pop*size(change_weight,2)/(neurons*neurons-neurons-
1));%number of weights to be changed


%% Create Matrix
W_small=rand(neurons,neurons,pop);
for i=1:pop
    W_big(:,:,i)=randint(neurons,neurons,[lower,upper]);
end
W_square=W_small+W_big;
W_square(1,:,:)=0; W_square(neurons,neurons,:)=0;
W_init=W_square;
B_small=rand(1,neurons,pop);
for i=1:pop
    B_big(:,:,i)=randint(1,neurons,[lower,upper]);
end
B_init=B_small+B_big;
B=B_init;
% create exact matrix
W_small=zeros(neurons,neurons,pop);
B_small=zeros(1,neurons,pop);
if rem(upper,1)==0
    for i=1:pop
        W_big(:,:,i)=randint(neurons,neurons,[lower,upper]);
        B_big(:,:,i)=randint(1,neurons,[lower,upper]);
```

```
        for j=1:neurons
            for k=1:neurons
                W_small(j,k,i)=rand(1)*(-1)^randint;
                B_small(1,k,i)=rand(1)*(-1)^randint;
            end
        end
    end
else
    W_big=zeros(neurons,neurons,pop);
    B_big=zeros(1,neurons,pop);
    for i=1:pop
        for j=1:neurons
            for k=1:neurons
                W_small(j,k,i)=0.5*rand(1)*(-1)^randint;
                B_small(1,k,i)=0.5*rand(1)*(-1)^randint;
            end
        end
    end
end
W_interval=W_small+W_big;
W_interval(1,:,:)=0;
W_interval(neurons,neurons,:)=0;
% initial intervals for bias
B_init=B_small+B_big; % for ~all case, B_init=zeros(1,neurons,pop)
B=B_init;

%% create initial weight matrix
W_exact=[0,0,0,0,0,0;
    0,0,-1,0,0,0;
    1,0,0,0,0,0;
    0.020697585281717,-0.019164430816405,0,0,0.958221540820238,-
1.954771943273285;
    0,0,0,0,0,-1;
    0,0,0,1,0,0;];
for i=1:pop
    W_square(:,:,i)=W_exact+W_interval(:,:,i);
end
for i=1:pop
```

```
    [W(i,:)]=square2string(neurons,W_square(:,:,i),B(:,:,i));
end
W_init=W;

% calculate output and weights for initial weight matrices
for i=1:pop
    [y(i,:),E(i,:)]=dynamic_rgnn(x,yd,neurons,W(i,:));
end
y_init=y;
E_init=E;
% initiate genetic algorithm optimization
count=1;

%% Genetic Training
% Note: next if loop is to ensure that network never gets worse.
while
count<max_count&&(min(E(:,1))>0.02||isnan(sum(E(:,1)))==1)&&toc<max_tim
e
    if size(change_weight,2)==neurons*neurons-neurons-1
        change_pop=[1:pop]';
    else
        change_pop=zeros(w_change,1);
        change_pop(1)=randint(1,1,[1,pop]);
        for i=2:w_change
            change_pop(i)=randint(1,1,[1,pop]);
            for j=1:i-1
                while change_pop(i)==change_pop(j)
                    change_pop(i)=randint(1,1,[1,pop]);
                end
            end
        end
    end

[W_cross,y_cross,E_cross]=genetic_floating_dynamic_newmut(E,W,pop,x,yd,
neurons,count,max_count,upper,lower,change_pop,change_weight);
    for i=1:pop
        if E_cross(i,1)==inf&&E(i,1)==inf
            if E_cross(i,2)<E(i,2)
```

```
                E_cross(i,:)=E(i,:);
                W_cross(i,:)=W(i,:);
                y_cross(i,:)=y(i,:);
            end
        elseif E_cross(i,1)<inf&&E(i,1)<inf
            if E_cross(i,1)>E(i,1)
                E_cross(i,:)=E(i,:);
                W_cross(i,:)=W(i,:);
                y_cross(i,:)=y(i,:);
            end
        elseif E_cross(i,1)==inf&&E(i,1)<inf
            E_cross(i,:)=E(i,:);
            W_cross(i,:)=W(i,:);
            y_cross(i,:)=y(i,:);
        end
    end
    W=W_cross;
    y=y_cross;
    E=E_cross;
    [ind,ind_h,ind_l]=max_min(E,pop);
    if rem(count,1)==0
        count
        ind_l
        ind_h
    end

data_out(:,count)=[count;ind_h(1,2);ind_h(1,3);ind_l(1,2);ind_l(1,3);to
c];
    data_E(:,((count-1)*2+1):((count-1)*2+2))=E;
    count=count+1;
end
ind=[[1:pop]' E];
ind_min=ind(1,:);
for i=2:pop
    if ind(i,2)<ind_min(1,2)
        ind_min(1,:)=ind(i,:);
    end
end
```

```
W_min=W(ind_min(1,1),:);
y_min=y(ind_min(1,1),:);
training_time=toc;
count
min(E(:,1))
save('genetic_step_W_1_july18')
```

C.3.1 Crossover and Mutation Algorithm (`genetic_floating_dynamic_newmut.m`)

The following function calculates a new set of individuals based upon the pervious population.

Input(s):

- Previous population characteristics (`E, W`)
- RGNN population characteristics (`pop, neurons`)
- Training data (`x, yd`)
- Number of times crossover and mutation have occurred (`count`)
- The maximum number of times crossover and mutation can occur (`max_count`)
- Mutation Characteristics (`upper, lower, change_pop, change_weight`)

Output(s):

- Weights after crossover and mutation (`W_cross`)
- Output of RGNNs after crossover and mutation (`y_cross`)
- Error or RGNNs after crossover and mutation (`E_cross`)

```
function
[W_cross,y_cross,E_cross]=genetic_floating_dynamic_newmut(E,W,pop,x,yd,
neurons,count,max_count,upper,lower,change_pop,change_weight)

% Calculate fitness of each individual
[fit_ind]=genetic_fitness(E,pop);
```

```
% Find Most Fit Individual
max_fit=[0 0];
for i=1:pop
    if fit_ind(i)>max_fit(1,2)
        max_fit=[i fit_ind(i)];
    end
end

%Create probability matrix
ind_range=zeros(pop,1);
ind_range(1)=fit_ind(1);
for i=2:pop
    ind_range(i)=ind_range(i-1)+fit_ind(i);
end
ind_rand=zeros(pop,size(W,2));
for i=1:pop
    for j=1:size(W,2)
        ind_rand(i,j)=rand(1);
    end
end

%Create weight index matrix to reference population Weights
w_index=zeros(pop,size(W,2));
for i=1:pop
    for j=1:size(W,2)
        for k=1:pop
            if k==1
                if ind_rand(i,j)<ind_range(k)
                    w_index(i,j)=k;
                end
            end
            if k>1
                if ind_rand(i,j)>=ind_range(k-
1)&&ind_rand(i,j)<ind_range(k)
                    w_index(i,j)=k;
                end
            end
```

```
        end
        if w_index(i,j)==0
            w_index(i,j)=max_fit(1,1);
        end
    end
end

%create new weight matrix
W_update=zeros(pop,size(W,2));
for i=1:pop
    for j=1:size(W,2)
        W_update(i,j)=W(w_index(i,j),j);
    end
end
W_2cross=W_update;

%mating
ind_mate=zeros(pop,3);
ind_mate(:,1)=[1:pop]';
for i=1:pop
    if ind_mate(i,2)==0
        ind_mate(i,2)=randint(1,1,[1,pop]);
        while
ind_mate(i,2)==ind_mate(i,1)||ind_mate(ind_mate(i,2),2)~=0
            ind_mate(i,2)=randint(1,1,[1,pop]);
        end
        ind_mate(ind_mate(i,2),2)=ind_mate(i,1);
    end
end


%floating-point crossover [Mahanti,2005]
for i=1:pop

[y_2cross(i,:),E_2cross(i,:)]=dynamic_rgnn(x,yd,neurons,W_2cross(i,:));
end
[W_cross,ind_mate]=genetic_crossover(W_2cross,E_2cross,ind_mate);
```

```
for i=1:pop
   [y_cross(i,:),E_cross(i,:)]=dynamic_rgnn(x,yd,neurons,W_cross(i,:));
end

for i=1:pop
    if isinf(E_cross(i,1))==1&&isinf(E_2cross(i,1))==0
        ind_mate(i,3)=0;
        ind_mate(ind_mate(i,2),3)=0;
    elseif isinf(E_cross(i,1))==1&&isinf(E_2cross(i,1))==1
        if E_cross(i,2)<E_2cross(i,2)
            ind_mate(i,3)=0;
            ind_mate(ind_mate(i,2),3)=0;
        end
    elseif isinf(E_cross(i,1))==0&&isinf(E_2cross(i,1))==0
        if E_cross(i,1)>E_2cross(i,1)
            ind_mate(i,3)=0;
            ind_mate(ind_mate(i,2),3)=0;
        end
    end
end

count_parent=1;
while count_parent<10&&min(ind_mate(:,3))==0
    [W_cross,ind_mate]=genetic_crossover(W_2cross,E_2cross,ind_mate);
    for i=1:pop

[y_cross(i,:),E_cross(i,:)]=dynamic_rgnn(x,yd,neurons,W_cross(i,:));
    end
    for i=1:pop
        if isinf(E_cross(i,1))==1&&isinf(E_2cross(i,1))==0
            ind_mate(i,3)=0;
            ind_mate(ind_mate(i,2),3)=0;
        elseif isinf(E_cross(i,1))==1&&isinf(E_2cross(i,1))==1
            if E_cross(i,2)<E_2cross(i,2)
                ind_mate(i,3)=0;
                ind_mate(ind_mate(i,2),3)=0;
            end
        elseif isinf(E_cross(i,1))==0&&isinf(E_2cross(i,1))==0
```

```
            if E_cross(i,1)>E_2cross(i,1)
                ind_mate(i,3)=0;
                ind_mate(ind_mate(i,2),3)=0;
            end
        end
    end
    count_parent=count_parent+1;
end

%% Mutation (modified from Mahanti [2005])
for i=1:size(change_pop,1)
    r1=rand;
    r2=rand;
    r3=rand;
    r4=change_weight(1,randint(1,1,[1,size(change_weight,2)]));
    b=1;
    f=(r2*(1-(count/max_count)))^b;
    if r1<0.5
        W_cross(change_pop(i),r4)=W_cross(change_pop(i),r4)+r3*upper*f;
    else
        W_cross(change_pop(i),r4)=W_cross(change_pop(i),r4)-r3*lower*f;
    end
end

%recalculate output and error
for i=1:pop

[y_cross(i,:),E_cross(i,:)]=dynamic_rgnn(x,yd,neurons,W_cross(i,:));
end
```

C.3.2 Genetic Fitness (`genetic_fitness.m`)

The purpose of the following function is to calculate the fitness of each individual in the population.

Input(s):

- Error matrix containing the error of each individual in the population (`E`)
- The number of individuals in the population (`pop`)

Output(s):

- The fitness range of each individual in the population (`fit_ind`)

```
function [fit_ind]=genetic_fitness(E,pop)

E_inf=zeros(size(E));
E_real=zeros(size(E));
for i=1:pop
    if E(i,1)==inf||isnan(E(i,1))==1
        E_inf(i,:)=E(i,:);
    else
        E_real(i,:)=E(i,:);
    end
end
E_r_total=sum(E_real(:,1));
E_i_total=sum(E_inf(:,2));

fit_real=zeros(pop,1);
fit_inf=zeros(pop,1);
for i=1:pop
    if isinf(E(i,1))==0&&isnan(E(i,1))==0
        fit_real(i)=(E_r_total-E_real(i,1))/E_r_total;
    else
        fit_inf(i)=E_inf(i,2)/E_i_total;
    end
end

fit_real_temp=fit_real;
for i=1:pop
    if fit_real(i)==0
        fit_real_temp(i)=inf;
    end
end
if min(fit_real_temp)~=inf&&max(E(:,1))==inf
```

```
    min_fit_real=min(fit_real_temp);
    max_fit_inf=max(fit_inf);
    adjust=(min_fit_real/max_fit_inf)/5;
    fit_inf=fit_inf*adjust;
end

fit=fit_real+fit_inf;
fit_total=sum(fit);

fit_ind=fit/fit_total;
```

C.3.3 Genetic Crossover (`genetic_crossover.m`)

The purpose of the following is to take in a set of weights - along with their error, and mating pairings - then output a crossed over set of weights.

Input(s):

Weights of every individual in the population (`W`)

Error of the individuals in the population (`E`)

The fitness range of each individual in the population (`fit_ind`)

Output(s):

Weights of population after crossover (`W_c`)

Index of weights which were mated (`ind_mate`)

```
function [W_c,ind_mate]=genetic_crossover(W,E,ind_mate)

%% The program starts here
pop=size(E,1);
W_c=W;
for i=1:pop
    if ind_mate(i,3)==0
        if isinf(E(i,1))==0&&isinf(E(ind_mate(i,2),1))==1
```

```
            W_c(ind_mate(i,2),:)=W_c(i,:)+rand(1)*(W_c(i,:)-
W_c(ind_mate(i,2),:));
            ind_mate(i,3)=1;
            ind_mate(ind_mate(i,2),3)=1;
        elseif isinf(E(i,1))==1&&isinf(E(ind_mate(i,2),1))==0

W_c(i,:)=W_c(ind_mate(i,2),:)+rand(1)*(W_c(ind_mate(i,2),:)-W_c(i,:));
            ind_mate(i,3)=1;
            ind_mate(ind_mate(i,2),3)=1;
        elseif isinf(E(i,1))==1&&isinf(E(ind_mate(i,2),1))==1
            if E(i,2)>E(ind_mate(i,2),2)
                W_c(ind_mate(i,2),:)=W_c(i,:)+rand(1)*(W_c(i,:)-
W_c(ind_mate(i,2),:));
                ind_mate(i,3)=1;
                ind_mate(ind_mate(i,2),3)=1;
            else

W_c(i,:)=W_c(ind_mate(i,2),:)+rand(1)*(W_c(ind_mate(i,2),:)-W_c(i,:));
                ind_mate(i,3)=1;
                ind_mate(ind_mate(i,2),3)=1;
            end
        elseif isinf(E(i,1))==0&&isinf(E(ind_mate(i,2),1))==0
            if E(i,1)<E(ind_mate(i,2),1)
                W_c(ind_mate(i,2),:)=W_c(i,:)+rand(1)*(W_c(i,:)-
W_c(ind_mate(i,2),:));
                ind_mate(i,3)=1;
                ind_mate(ind_mate(i,2),3)=1;
            else

W_c(i,:)=W_c(ind_mate(i,2),:)+rand(1)*(W_c(ind_mate(i,2),:)-W_c(i,:));
                ind_mate(i,3)=1;
                ind_mate(ind_mate(i,2),3)=1;
            end
        end
    end
end
```